CRitical S**YST**em Engineering **A**cce**L**eration

Use – Case Definition

D304.011

DOCUMENT INFORMATION

Project	CRYSTAL
Grant Agreement No.	ARTEMIS-2012-1-332830
Deliverable Title	Milestone Report – V1
Deliverable No.	D304.011
Dissemination Level	CO
Nature	R
Document Version	V1.0
Date	2014-01-30
Contact	Gerald Stieglbauer
Organization	AVL
Phone	+43 316 787 7803
E-Mail	gerald.stieglbauer@avl.com

AUTHORS TABLE

Name	Company	E-Mail
Gerald Stieglbauer	AVL	gerald.stieglbauer@avl.com
Jörg Settelmeier	AVL-R	joerg.settelmeier@avl.com

REVIEW TABLE

Version	Date	Reviewer
0.65	15.01.2014	Jörg Settelmeier (Common and UC3.4a part)
0.7	16.01.2014	Gerald Stieglbauer (UC3.4b part)
0.9	22.01.2014	Reuter/Hopp/Melzi External Review
1.0	28.1.2014	Adaptation according to external review comments

CHANGE HISTORY

Version	Date	Reason for Change	Pages Affected
0.1	1.12.2013	Initial Versions of UC3.4a and UC3.4b descriptions	
0.5	11.12.2013	Merge of UC description in a single document	
0.6	10.01.2014	Internal Review of JoSe – Chapter 1 and 2	1-12
0.61	10.01.2014	Internal Review of JoSe – Chapter 3	13-17
0.62	10.01.2014	Internal Review of JoSe – Chapter 5, 5.1	26-30
0.63	13.01.2014	Internal Review of JoSe – Chapter 5, 5.2	
0.64	13.01.2014	Internal Review of JoSe – Chapter 7	
0.65	15.01.2014	Internal Review of JoSe of remaining chapters	
0.7	16.01.2014	Internal Review of GST	
0.8	16.01.2014	Integration of review results into document – ready for external review	
0.9	23.01.2014	External Reviewed Document	
1.0	28.01.2014	Integration of review results	

CONTENT

Content of Figures

Content of Tables

1 Introduction

1.1 Role of deliverable

This document has the following major purposes:

- Define of the overall use case, including a detailed description of the underlying development processes and the set of involved process activities and engineering methods
- Provide input to WP601 (IOS Development) required to derive specific IOS-related requirements
- Establish the technology baseline with respect to the use-case, and the expected progress beyond (existing functionalities vs. functionalities that are expected to be developed in CRYSTAL)

1.2 Relationship to other CRYSTAL Documents

This document has a series of relationships especially to SP6 deliverables, whose related tools and methods should be improved, extended and applied to the use cases of WP3.4.
These relationships are:

- WP6.3 (System analysis and exploration)
    - Method of model-based requirement engineering (T6.10.3),
    - ARTISAN Studio integration (T6.3.15),
    - AVL Cruise integration (T6.3.5)
- WP6.5 (AUTOSAR Tools & Components)
    - Model Based Software Development (T6.5.1)
    - TTEthernet Design & Development Tools (T6.5.8)
    - SoS Infrastructure Middleware Bricks (T6.5.9)
    - SoS Infrastructure Wireless Interface (T6.5.10)
- WP6.8
    - HP QC integration (T6.8.4)
    - PTC Source und Integrity (workflow management) (T6.8.9)
- WP6.10
    - System Family Engineering Framework (T6.10.2)
    - Variant Analysis (T6.10.3)
    - automatic testing (SiL, HiL) (T6.10.4)
    - AVL Creta/Cameo integration (T6.10.7)
- WP6.11
    - Via PTC indirect relation to Communicate the IOS to SCLC (T6.11.2)
- WP6.12
    - AVL VeVaT/Magic integration and development (T6.12.5)
    - Integrated Tool Environment for embedded controls development former Embedded Verification Platform (T6.12.10 / T6.12.6)
- WP6.13
    - AVL Simulation Model Data Backbone (T6.13.1)
    - MathWorks Simulink (T6.13.2)
    - AVL ARTE.Lab (T6.13.4)

1.3 Structure of this document

The structure of the document is extended to the common UC template due to the reason that WP3.4 consists of two sub use case, i.e. UC3.4a and UC3.4b. The document is structured so that every chapter (except the use case overview Chapter 2 as well as Chapter 7, which is about the commonalities of the sub use cases) is described in a separate chapter. In that manner UC3.4a is described by the Chapters 3, 4 and 8, whereas UC3.4b is described by the Chapters 4, 6 and 9.

2 Use Case Overview and Motivation

AVL supports the automotive industry in many ways. On the one hand AVL helps vehicle manufacturers with engineering services to master their developing challenges. On the other hand, AVL creates and supplies also the products that support these engineering services. Products are for instances test benches, measurement devices but also software tools to handle these devices and testing processes. Even not stopping here, AVL goes also beyond that by a high degree of virtualization of testing methods and procedures through the use of simulation models for both, the unit under test as well as the testing environment: Tools and models for simulating a vehicle (or parts of it such as the combustion process) and its environment thus enables vehicle development frontloading. Development frontloading denotes the process of vehicle development, whereas no physical components are needed (or are partially replaced by simulation models) at an early development phase.

In the Crystal project, AVL participates with two use cases (UCs) reflected by the WP3.3 and WP3.4. This chapter is all about WP3.4. It consists of several sub-use cases, which are, however, related to each other and have partially overlapping content. Figure 2-1 shows an overview about this structure.

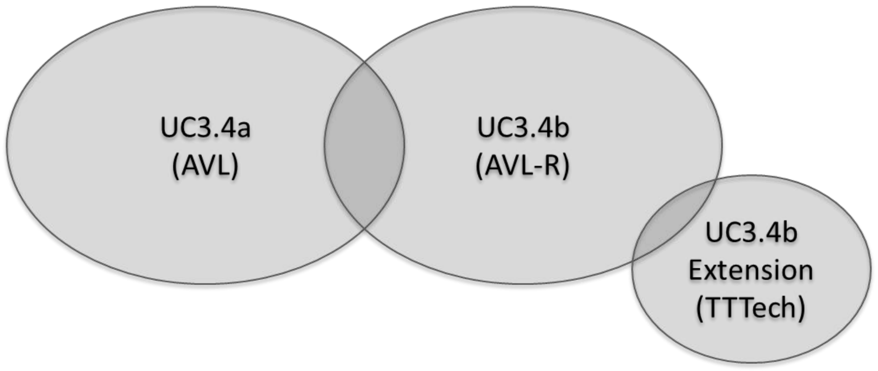


Figure 2-1 Overview of the use case structure

UC3.4a is led by AVL Graz and is about the interoperability challenges for vehicle testing scenarios at different vehicle development stages. AVL Regensburg is leading UC3.4b which is mostly about integrated tool environments for embedded control development and improvement of the development via an efficient variant handling within the V-cycle. The latter UC is extended by an additional aspect about the integration and configuration of versatile systems of systems (SoS) platforms, which is driven by TTTech.

In the following, the motivations of the two use cases are described as well as how these UCs are roughly related. The latter aspect will be explained in more detail in Chapter 7. Since both use case variants are heavily based on simulation models, whereas the same, similar or related models are applied in both use cases, a concept for simulation model exchange across the UCs has to be provided. This concept will be denoted as data backbone or simulation model data backbone (through its focus on development frontloading as mentioned above).

2.1 Overall Motivation of UC3.4a

UC3.4a deals with the complexity of different testing scenarios over the various vehicle development phases and the question how to enable an integrated development cycle over these development phases. To evaluate and improve the related interoperability challenges, a particular use case scenario about testing a hybrid vehicle under consideration of emission legislation is defined. Such an integrated development cycle should support answering specific questions regarding testing issues. These questions are for instance (by no means of being complete):

- Which concrete data artefacts were involved in a concrete test run performed three weeks ago?

- How can I re-use simulation models, measurement and calibration results in a later development phase?
- Are there differences (e.g. measurement results) for a concrete test run in different development phases (simulation phase vs. component test bed)?
- Why did I set up a certain test run as I did (e.g. artefact navigation)?
- What was the history of changes until I finally got my last test run results (e.g. version management)?

In corresponding testing environments a number of tools are involved, which are currently operating on many data sources. Answering question such as above lead to the strong need of tool interoperability on the one hand and some kind of central data repository on the other hand. UC3.4a will address both aspects.

2.2 Overall Motivation of UC3.4b

AVL Powertrain Engineering is a key development partner for innovative powertrain systems. From diesel engines to electric drives, from alternative fuels to control software, from transmissions to batteries, we have been supporting the automotive and mobility industries for more than 60 years. Unique synergies with AVL Instrumentation & Test Systems and AVL Advanced Simulation Technologies lead to highly creative, mature and application-specific solutions for our customers to meet their future market challenges.
To fulfil the Customer request, AVL-R follows the Software development process represented via the classical V-model. This V-model demonstrates the relationships between each phase of the development life cycle (left side) and its associated phase of testing (right part). It illustrates a general overview of development process but still excludes details such as chronological order, development iterations, etc.

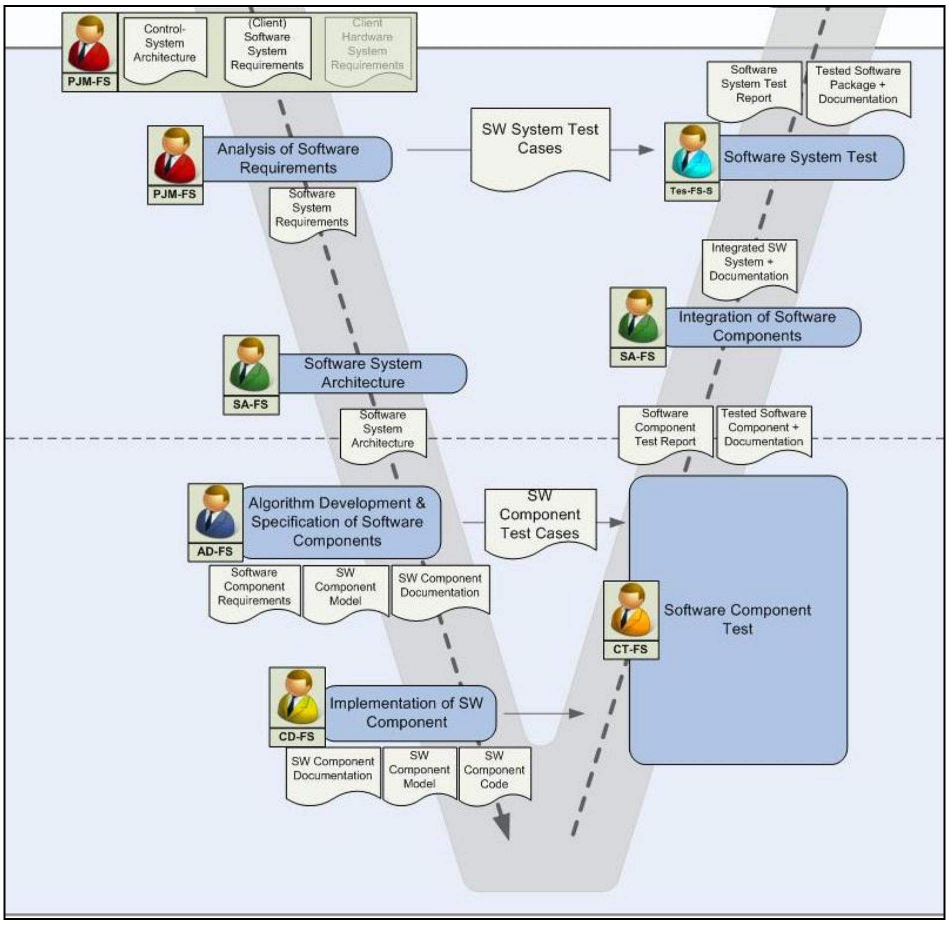

PJM-FS
Control-System Architecture
(Client) Software System Requirements
Client Hardware System Requirements
Software System Test Report
Tested Software Package + Documentation
PJM-FS
Analysis of Software Requirements
SW System Test Cases
Tes-FS-S
Software System Test
Software System Requirements
Integrated SW System + Documentation
SA-FS
Integration of Software Components
SA-FS
Software System Architecture
Software System Architecture
Software Component Test Report
Tested Software Component + Documentation
AD-FS
Algorithm Development & Specification of Software Components
SW Component Test Cases
Software Component Requirements
SW Component Model
SW Component Documentation
CT-FS
Software Component Test
CD-FS
Implementation of SW Component
SW Component Documentation
SW Component Model
SW Component Code

Figure 2-2: The FASD (**F**unction **A**nd **S**oftware **D**evelopment process) V-Model

On the left side of the V-model, the development process starts with the activity *analysis of software system requirements*. These *software system requirements* delivered by the customer are analysed and put into a form to be used as basis for the software development. Based on the *software system requirements*, the *software system test cases* and the *software system architecture and its software components* are defined. The *software system requirements* are assigned to the *software components* and handed over the function algorithm developers and software algorithm developers for further analysis.
Software component development starts with algorithm development and specification of the detailed *software component requirements*. This step already involves the set-up of a model itself (e.g. Simulink model) and is documentation. Based on the *software component requirements*, the *software component* is developed. After this development step, the software components adhere to the functionality specified in the software requirements.
On the right side of the V-model, the implemented software components have to be tested according to the defined test cases. As result of the defined test cases, *software component test reports* and the *tested software component* (incl. documentation) have to be delivered.
If all software components have been tested, they are integrated and validated with the final *software system tests*. Finally, the software system including the test results is delivered to the customer.

The main motivation for UC3.4b is to increase quality and efficiency of powertrain system engineering activities by implementation of a seamless tool chain, increased automation of HiL test case generation and a suitable simulation model variant management. On one side many steps are done manually, e.g. creating simulation models for different purposes, adapting them for several variants, compiling and composing them for a certain test case scenario and on the other side requirements are either defined implicitly within the models or are stored in a requirement management tool without an suitable relation to the respective models and tests. Due to this, an automatic verification is impossible which has to be clarified by CRYSTAL.

A major objective of this work package is to provide an integrated variant management and test case generation. This shall be done by coupling the appropriate tool chains within the CRYSTAL project and apply corresponding methods to ensure these major objectives. Due to an improved interoperability of modelling/simulation tools, the current situation can be improved significantly. Modelling/simulation environments could exchange or link data by applying well defined interfaces and services. Furthermore, a System-of-Systems platform will be linked for prototype testing of selected use case aspects.
Consequences are an improved degree of automation, a reduced set of development overhead and the Project collaboration becomes more straightforward.

3 Use Case Process Description of UC3.4a

This chapter describes the use case in a high level manner. This description is aligned with the definition of a so-called use case scenario. This scenario is the foundation in respect of creating a concrete prototype implementation later on during the CRYSTAL project. The scenario is comprises of some initial vehicle and testing requirements, which should be verified during an integrated (model-based) requirement engineering. The Chapters 4 and 8 – about UC details and engineering methods – are referring to this use case scenario where applicable.

3.1 The use case scenario for UC3.4a

The public use case for the automotive domain of the CRYSTAL (developed in WP3.7) is based on the overall user story of _building a vehicle_. _UC3.4a_ is related to this public use case *by applying AVL's methods, devices and tools throughout the development phases in form of testing procedures and techniques*. In particular, the use case is comprised of a particular _use case scenario_ (do not mistake this for _user story_ as defined in the CRYSTAL application document) about developing a _hybrid vehicle_. In addition, the use case is aligned with the principles of *integrated* _model-based requirement engineering_ (defined as _user story_ in the CRYSTAL application document).

In addition, the scenario should be applied on the latest trends regarding *standardized vehicle test cycles* and *emission legislation*, which will play even a more important role in the immediate future for vehicle manufactures than ever before. Consequently, the upcoming _WLTP_ (World-wide harmonized Light-duty Test Procedure) standard is an essential element in this use case.

As described in more detail by the following sections, *development frontloading* by early vehicle *simulation* and *calibration iterations* as well as re-using of these simulation and calibration results in *later development phase*s are further key challenges of this use case. All challenges have in common that corresponding methods and infrastructure for a sophisticated simulation model and data management has to be applied in order to establish *traceability* between all essential data artefacts.

3.1.1 Integrated Model-based Requirement Engineering as _user story_

The _user story_ of UC3.4a is about full integrated *(model-based) requirement engineering* throughout the whole development process. This means that in case of *building a vehicle,* it has to be shown (semi-automatically) that a vehicle or sub-parts of it (virtualized by simulation or not) are fulfilling the given requirements. This is only possible, if traceability is established between all artefacts that are created during the overall goal of *building a vehicle*. This traceability is needed to verify the given requirements and the requirements have to be (semi-)formalized by adequate requirements models in order to enable (semi-) automatic requirement verification. It is a research goal of this use case to compare (or even combine) several requirement formalization approaches (e.g. boilerplate approach, sequence charts, etc.).

The requirements itself will be derived from the WLTP standard on the one hand and typical additional (hybrid) vehicle requirements. An initial set of such requirements is presented later on in this chapter and will be enhanced and adapted during the CRYSTAL project.

3.1.2 Short Introduction to WLTP

Currently, various emission legislations are actively applied all over the world. In Europe, for instance, the Euro 5 is the latest released. Its equivalent in the US is Tier II or PNLT in Japan. Consequently, there is some need for harmonizing all these different approaches, which is one major goal of the WLTP. The project is set up accordingly to the UNECE 1998 world harmonization agreement. Originally sponsored by EU, Japan and the US, the latter one, however, has withdrawn sponsoring in June 2010. Other objectives besides just emission harmonization are:

- Development of a drive cycle that reflects an average world-wide driving behaviour.
- Development of harmonized test procedures by harmonized parameter definition, calculation formulas, testing equipment, etc.

The WLTP[1] (currently available as a draft) consists of two definition phases. In Phase I the drive cycle and test procedure standards are defined. In Phase II topics such as off-cycle emissions, low ambient temperature, high altitude, etc. will play an important role.

3.1.3 Interoperability challenges in UC3.4a

In the CRYSTAL project and thus also in UC3.4a, a major interoperability challenge is not to build (or standardize) every interoperability technique from scratch, but to re-use existing technologies and standards. This, however, does not mean that these existing standards should be harmonized to a single one. This would not be possible either due to the huge standardization efforts on the one hand, and on the other hand due to the fact that many standards a very specialized for a specific issue. Alternatively, existing standards should be used side-by-side and combined where possible.

Since in this use case establishing traceability across the several vehicle development phases, methods and disciplines should be established, there is a strong need to interlink the involved data artefacts (such as requirements, test cases, measurement results, etc.) across these development phases. For this data artefacts linkage, an additional interoperability approach should be applied that could operate independently from existing standards. The OSLC standard seems to be a good candidate for this demand. In the Figure 3-1 below, a generic pattern of using OSLC in common with other, more specialized standards, is illustrated.

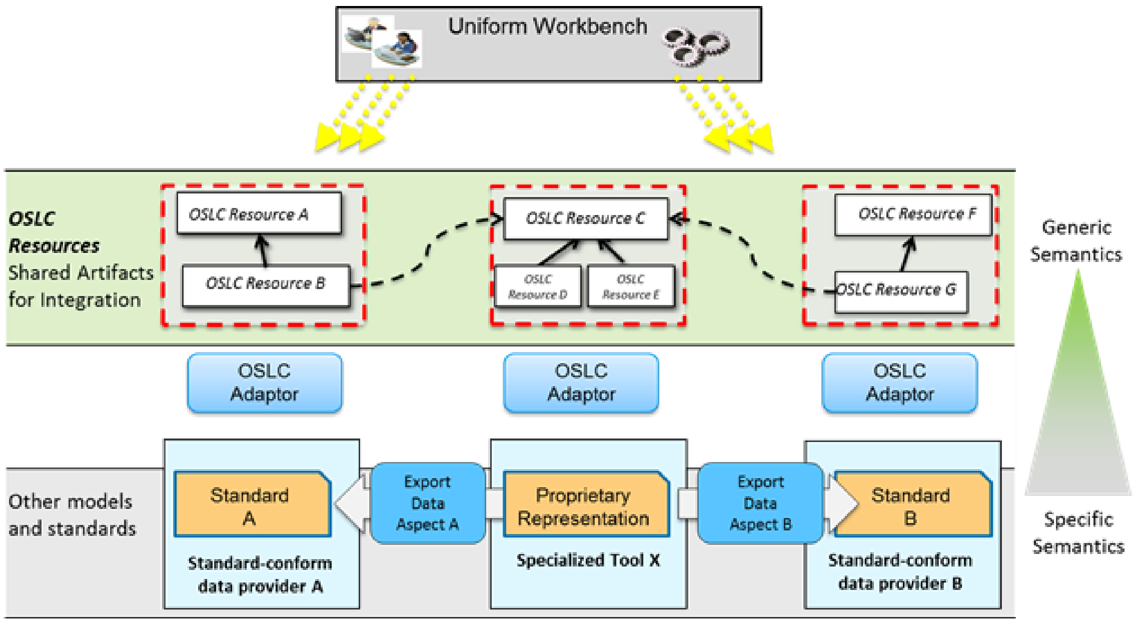


Figure 3-1: A generic pattern about applying OSLC to combine various standards

Bottom part of Figure 3-1:

A specialized tool X may save its data in a proprietary format containing all the details needed by the tool. In order to exchange the data with another tool for certain purpose Standard A was developed. Via an export function or using a tool interface (not necessarily standardized), Tool A is able to convert the data to this standard either by transform the full or only a part of its current data set. For another purpose, Standard B is about other aspects of the proprietary data set. Another export functions is also capable to perform that kind of transformation. In any case of export, data dependency between the proprietary and the exported data is usually not linked, which means that important information is lost.

[1] For more information see at https://www2.unece.org/wiki/pages/viewpage.action?pageId=2523179

Top part of Figure 3-1:

To compensate this loss and even go further, OSLC adds a kind of additional layer on top of the involved tools and data artefacts as illustrated by the figure above. So-called OSLC adapters map essential elements of the proprietary and the standardized data sets to so-called OSLC resources. OSLC resources are standardized elements assigned to a certain domain. A domain, for instance, could be Requirement Engineering (abbreviated with OSLC RM) or Quality Measurement (OSLC QM). The key idea behind these domains is that not every detail of the proprietary or standardized data set has to be mapped to an OSLC resource. Instead, a resource is only introduced if there is a linkage need across the OSLC domains, which significantly reduces complexity of creating an overall standard for interoperability: The complexity remains at the tools and OSLC just focus on the cross-dependency aspects of the data.

If a specialized standard (such as ASAM ODS for measurement results) goes beyond that cross-dependency aspect, then such an OSLC approach abstracts from the selected standard.. The same is true if there is a lack of a low level standard and only proprietary data representations are available. Finally, a uniform workbench would then be able to navigate on the top level OSLC elements but would still delegate to the tools and data providers for detailed data operations and representation visualizations.

The interoperability challenges of UC3.4a are mostly derived from the interoperability pattern described above represented by corresponding *engineering methods* such as:

- *Formalize requirements*, e.g. the WLTP requirements
- *Verification of Requirements* through analysis of the formalized requirements
- Requirement Traceability to Authoring Tools
- *Test and calibration iteration* in the context of testing a vehicle
- Data management across development phases
- *Heterogeneous Simulation* of different kind of models

Each of this engineering method and its relation to the interoperability pattern above will be explained in more detail in Chapter 8.

3.2 Testing a hybrid vehicle according to WLTP as *use case scenario*

In our use case scenario, we want to highlight a particular aspect of the overall goal of *building a vehicle*, namely *testing a vehicle*. Even more specific we are focussing on *testing a hybrid vehicle* according to the specifications of the WLTP.

In order to describe the *use case scenario* in more detail, *selected requirements* are applied on a concrete testing procedure with concrete tools and artefacts. These requirements are by no means complete, but are used to describe the use case from a practical point of view. During the project the requirement list will be updated and enhanced accordingly. In addition, the requirements can either belong to *vehicle requirements* (see Section 3.2.1.2 - driven by certain example scenarios) or special *testing requirements* (see Section 3.2.1.3) as demanded by the WLTP or even both as illustrated in the Figure below by the overlapping area (see Section 3.2.1.1). Or in other words, for testing a vehicle, the original set of requirements is extended by those that are just needed for testing aspects (e.g. a certain configuration demands of a test bed, concrete test-run constraints, etc.).

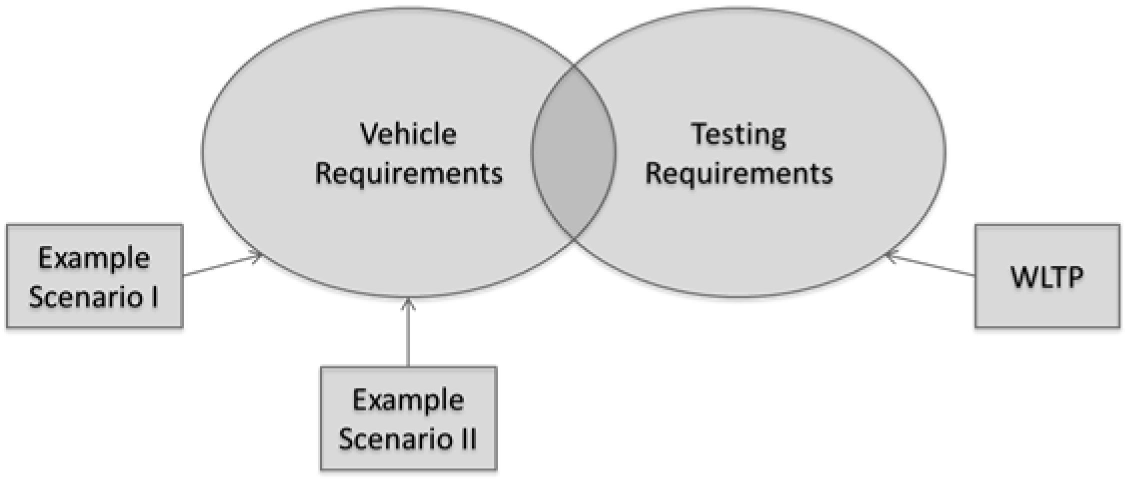


Figure 3-2: Vehicle vs. Testing requirements

The selected requirements are described in more detail in the following subsections. These requirements are the starting base for the overall development process with its phases; they are applied on the engineering methods and will be verified by executing concrete test-runs for a specific hybrid vehicle.

3.2.1.1 Emission limitation requirement

A requirement that belongs to both categorizations, i.e. a vehicle requirement and a testing requirement, is for instance a requirement called *emission limit requirement*: As defined in the WLTP specification, the CO2 emission of a certain vehicle should be below a certain level, let's assume 95g/km in our case. In order to enable requirement validation and verification, this requirement needs then to be formalized accordingly (e.g. by and appropriate requirement model) in the first step. Then, the requirement should be captured by every development step and has to be verified by appropriate testing procedures. For instance, the initial step would be to test various *architectural vehicle designs* (a hybrid vehicle design would be one of them) by simulation of appropriate *vehicle models* that represent these *architectural vehicle designs*.

During this simulation, the models have to be configured within their possible range by performing *parameterization* and *calibration testing iterations*.

Parameterization means to modify the physical specification of certain sub-elements of the virtual vehicle, whereas *calibration* means to modify a certain configuration of sub-elements, without the need of physically exchanging it. For instance, changing the value of the *cylinder capacity* or the *number of gears* of an automatic gear box belongs to *parameterization*, whereas changing the *injection time* of the engine or the *shifting thresholds* of the automatic gear box belongs to *calibration*. For some more examples see Table 1.

Simulating the models (i.e. testing the virtual vehicle) leads to *simulation results* that verify the given requirements for a concrete *parameterization* and *calibration* set (see Table below) or falsify the models in combination with their associated architectural vehicle design (such as a hybrid vehicle). In the latter case, the parameterization and calibration has to be modified and adjusted followed by further simulation test iteration(s).

In later development phases, simulation models will be more and more replaced by real-world components (e.g. engine, gearbox, etc.). However, the parameterization and calibration of the unit under test (i.e. the hybrid vehicle) remain the same in principle with the difference that pure architectural design parameters of the real-world component cannot be changed without a physical re-design (e.g. changing the number of cylinder capacity of an engine) as it is possible for virtual components of a simulation models. Or in other words: efforts of changes increases over the different development phases. A concrete list of development phases considered in this use case are listed and described in Chapter 4.

Description	Type	Typical value, e.g.
Cylinder capacity	Parameterization	1.3l
Number of gears	Parameterization	5
Injection time	Calibration	50ms
Shifting thresholds (for shift up)	Calibration	100 rad/s
Shifting thresholds (for shift down)	Calibration	400 rad/S

Table 1: Examples for parameter and calibration data

3.2.1.2 Vehicle requirements

Independent of the WLTP or any other emission specification, vehicle requirements (e.g. postulated by a specific customer) are considered during the vehicle development process. For our use case scenario, we are starting some vehicle requirements listed in Table 2, which will be extended accordingly during the project:

Requirement description	Example Value
Maximal speed	150 km/h
Time for acceleration 0 to 50 km/h	<= 5s
Range of vehicle	> 200 km
Fuel consumption	<4.5l/100km
Maximal Weight	1.5t
Number of Passengers	5

Table 2: Examples for vehicle requirements

3.2.1.3 Testing requirements

Finally, emission specifications such as the WLTP add further requirements on the testing procedure (input vector set), rules (constrains on the test run execution), methods (calculations, formulas) and equipment (such as measurement devices). In our use case scenario, we have a close look on three special gear shifting requirements (i.e. testing rules) as defined by the WLTP.

- *WLTP_RQ1*. First gear shall be selected 1 second before beginning an acceleration phase from standstill with the clutch disengaged. Vehicle speeds below 1 km/h imply that the vehicle is standing still.
- *WLTP_RQ2*. Gears shall not be skipped during acceleration phases. Gears used during accelerations and decelerations must be used for a period of at least 3 seconds.
- *WLTP_RQ3*. Gears may be skipped during deceleration phases. For the last phases of a deceleration to a stop, the clutch may be either disengaged or the gear lever placed in neutral and the clutch left engaged.

Note that these requirements do not have anything to do with the vehicle design. The vehicle itself may be still able to break with any of these rules (e.g. skipping a gear during acceleration), however during testing, they have to be ensured. Finally, these three testing requirements are formulated by ordinary prose. In order to enable (semi-)automatic verification of these requirements, they have to be formalized by an appropriate method. In this use case, several approaches are planned to be applied for requirement formalization.

4 Detailed Description of the Use Case Process of UC3.4a

This chapter explains the role and important aspects of testing a vehicle throughout the overall vehicle development process. For that purpose, so-called testing V-models enhance the general vehicle development V-model. The relations between these testing V-models and the classical vehicle development V-model are described below. Finally, interoperability challenges are derived from that.

4.1 Activities

As indicated in the Chapter 3, requirements verification through simulation models is only one out of other development phases. In this section, these development phases and their belonging activities are described in more detail by applying them on the *classical V-model* on the one hand and by introducing a *testing V-model* on the other hand.

4.1.1 Classical V/W-model aspects

Since several decades, the prominent V model (suggested by Barry Boehm in the year 1979) is present in the automotive industry and represents the typical stages of vehicle manufacturing and the validation of these processes. A V-model has typically a central *development entity*, whose creation, developing and manufacturing it is all about. In case of the automotive industry and according to our use case, this development entity is a *hybrid vehicle*. A typical V model as applied in the automotive industry is illustrated in Figure 4-1.

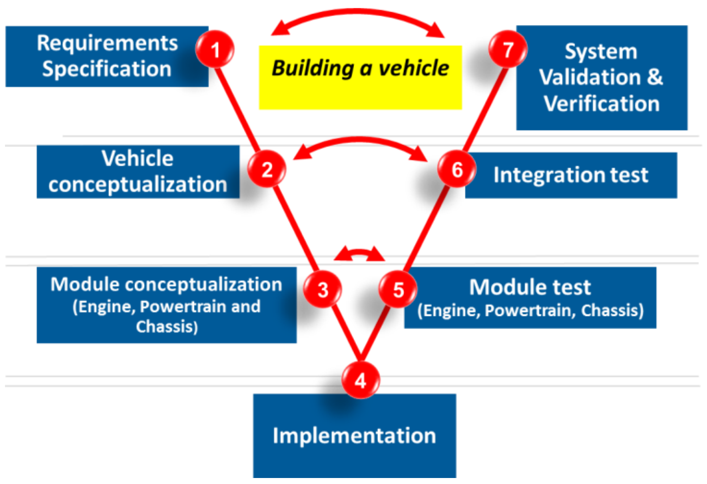

Requirements Specification
1
Building a vehicle
7
System Validation & Verification
Vehicle conceptualization
2
6
Integration test
Module conceptualization (Engine, Powertrain and Chassis)
3
5
Module test (Engine, Powertrain, Chassis)
4
Implementation

Figure 4-1: Classical V-model applied on the topic of building a vehicle

Based on a set of vehicle requirements, an engineer creates an overall vehicle concept (maybe based on previous projects with similar requirements). Then, a team of engineers continues with the conceptualization of the subparts (such as engine, powertrain, and chassis) followed by the development of concepts for even more details and so on. After the implementation of the concepts, first the modules are tested against their particular concept specification. Then the modules are integrated step-wise, which goes along corresponding integration tests. Finally, the original requirements of the resulting vehicle are verified.

Variants of this classical representation of a V-model add the additional aspect of virtualization of the implementation in form of simulation models (e.g. of an engine). These simulation models make validation of certain design decisions possible at an early stage - also called development frontloading. Development frontloading enables early comparison and validation of different designs. Independent of that, the goal of or the story behind these models is the same: Building a vehicle.

A reasonable enhancement of the V-model in that manner illustrated above in form of a so-called W-model[2] is illustrated in Figure 4-2.

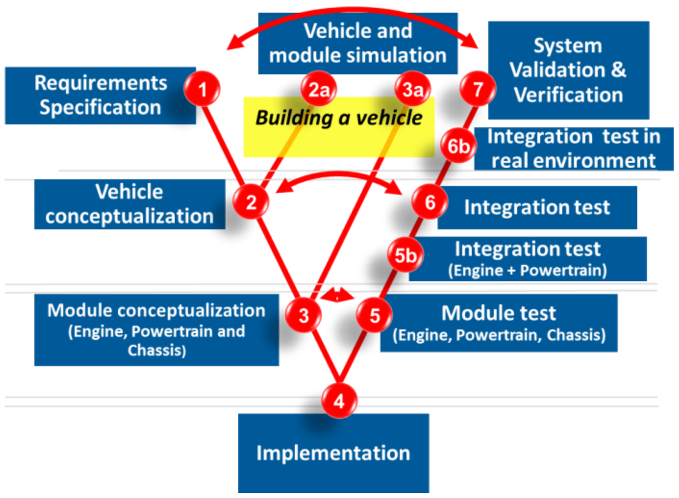

Vehicle and module simulation
System Validation & Verification
Requirements Specification
1
2a
3a
7
Building a vehicle
6b
Integration test in real environment
Vehicle conceptualization
2
6
Integration test
5b
Integration test (Engine + Powertrain)
Module conceptualization (Engine, Powertrain and Chassis)
3
5
Module test (Engine, Powertrain, Chassis)
4
Implementation

Figure 4-2: Early verification through frontloading W Model

As shown in this figure, requirements validation and verification becomes possible already at the *vehicle* or *module conceptualization* phase (2 and 3) by *vehicle and module simulation* (2a and 3a).

For simplicity, we distinguish by now between three elementary vehicle modules, namely *engine*, *powertrain,* and *chassis*. During module conceptualization these modules are specified through design, parameterization and calibration in an iterative simulation and test cycle. During these iterations, modules are specified at an ever more detailed level, e.g. by separating the engine model into two models, whereas the first one covers the physical aspects of the engine and the second one is a model of the engine control unit.

Through this stepwise refinement, the implementation phase (4) finally results in real physical components (engine, powertrain, chassis, control units, etc.) and software that runs on the control units (and may be automatically generated out of the simulation models).

These components and software units need to be tested in concrete module tests (5) according to the given requirements. In addition, intermediate integration tests may be added (5b), which combines for instance engine and powertrain testing but still includes a *rest vehicle simulation*, of those parts that are not physically present in the particular testing scenario. In both scenarios, additional equipment is needed to test the physical components, i.e. a so-called test bed. This test bed needs to be configured according to the vehicle requirements as well.

Development phase (6) leads then to a full integration test. This is as well accomplished with the use of a test bed, where the environment of the vehicle (driver, street, weather conditions, etc.) still needs to be simulated. In order to overcome this remaining simulation part as well, so-called in-vehicle tests are performed, where a real driver is steering the vehicle on a real road and the testing equipment is built directly into a car (6b). This final test phase leads then to the system validation and verification (7) according to the given requirements.

APPLYING THE *USE CASE SCENARIO* ON THE CLASSICAL V/W-MODEL

In our *use case scenario*, the *emission limit requirement* has to be tested in all conceptualization, development, implementation and integration phases. In order to achieve that, traceability of the *emission limit requirement* and the corresponding artefacts in the testing phases have to be established. Artefacts are

[2] The W-model presented in this document is loosely inspired by that proposed by Herzlich, P. (1993). "The Politics of Testing", proceedings of 1st EuroSTAR conference, London, Oct. 25-28, 1993. Herzlich suggest early verification of activities on the left leg of the V-model by introducing testing approaches.

in this case simulation models, parameterization and calibration data, simulation (or measurement) results, test-runs data and so on. Regarding our use case scenario of a hybrid vehicle we have thus simulation models about the combustion and the electrical engine including the control unit of those; we need a battery model as well as parameterization data (e.g. cylinder capacity of the combustion engine, weight of the battery, etc.) and calibration data (e.g. ignition time of the combustion engine, shifting time of the gear box, etc.). Certain test-runs defines at what level and at what time the accelerator and brake pedal is pushed, and so on. These test-runs lead to outputs in form of simulation or measurement results, respectively the test contain virtual components or not.

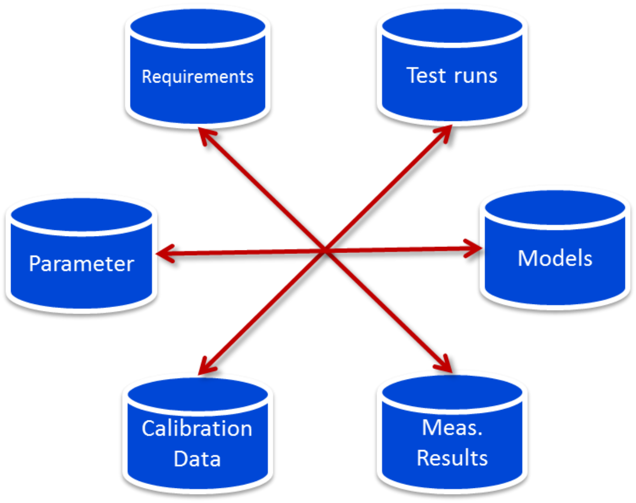


Figure 4-3 Schematic illustration of interlinking of data artefacts

As illustrated in the Figure 4-3 and in order to validate the *emission limit requirement*, all artefacts need to be linked to each other in a sufficient manner to associate this requirement with the final test results (e.g. the maximum of the occurred CO2 value during a particular test-run). But even beyond that, important questions (such as which test-run performed at what time is associated with what requirements, models, calibration and parameterization data) have to be answered. This would allow detailed (historical) analysis of a particular test phase and would even enable detailed comparison of different test phases (e.g. answering the question, if simulation leads to the same results as the real-world test or if it is necessary to improve the accuracy of my models).

4.1.2 Testing V-model aspects

The overall goal of the public use case in CRYSTAL's SP3 is about building a vehicle. Testing a vehicle is a part of the process of building a vehicle. Applying appropriate test methods, devices, and tools throughout the development processes is essential. Adequate testing procedures, techniques, and environments finally should verify the given vehicle requirements.

In addition to the given *vehicle requirements*, specific *testing requirements* enhances the list of requirements. These requirements are based for instance on a certain testing specifications such as the WLTP (see Section 3.1.2) or others. Such specifications define concrete test-runs, the accuracy of measurement devices, formulas that have to be applied for measurement result analysis, and so on. The needed test environment (needed measurement devices, test-run input vectors, etc.) is then derived from this testing requirement as well.

The concrete configuration of the test environment depends then on the concrete position within the V or W model for *building a vehicle*. For instance, for early simulation phases, simulation models (e.g. for control software and virtualized physical components such as engines, etc.) are fully sufficient, whereas after the implementation phase, various test bed settings are required for a stepwise substitution of simulation models by their physical counterparts. Furthermore, the selection of development and testing tools may differ as well, since one tool is more specialized for a certain development phase than another. Or in other words:

Every testing phase may come up with its own tool set. It is one key interoperability challenge to share data across these development and testing phases as described in more detail by Section 4.2. Setting up test environments is a complex task and needs to be easily adaptable to the requirements of the current position in the development process (i.e. the V/W-model) including addition boundaries (or testing requirements) coming from specifications such as the WLTP. Further enhancements are specifications to define concrete test-runs, the accuracy of measurement devices, formulas that have to be applied for measurement result analysis and so on.

This means that besides the vehicle and model conceptualization and implementation on the one hand (left side of the V-model) and the actual test case execution, system validation and verification on the other hand (right side of the V-model), a lot of knowledge and efforts are going into the test case conceptualization and implementation as well. One may say that this test case conceptualization and implementation is a further sub-development step of the overall assignment of building a vehicle. Depending on the concrete position in the V/W model, however, a particular test case conceptualization and implementation has its own selection of testing processes and techniques in form of testing tools and methods as stated above. Therefore, testing a vehicle is a development task by its own and can therefore be represented by a separate V-model. In contrast to a classical automotive V or W model that focuses on building a vehicle, an alternative V model that focus on testing a vehicle would look as illustrated in Figure 4-4.

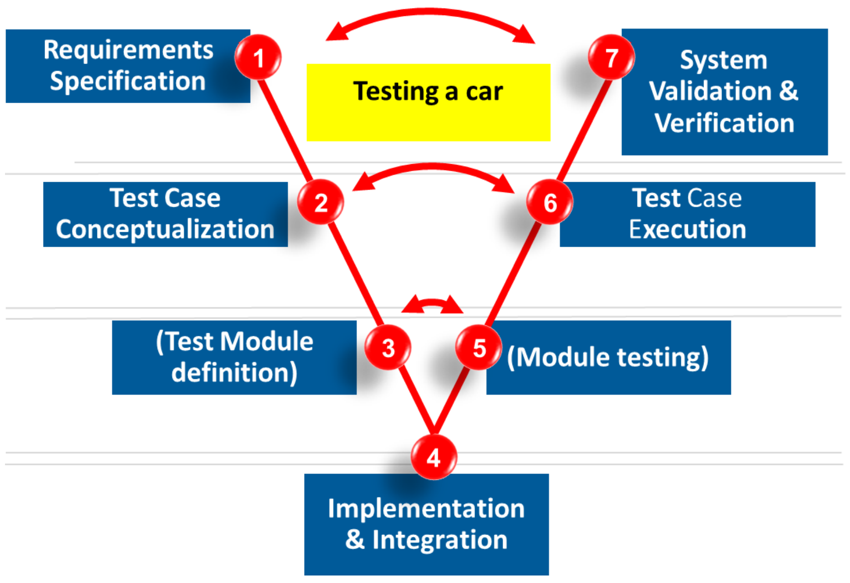

Requirements Specification
1
Testing a car
7
System Validation & Verification
Test Case Conceptualization
2
6
Test Case Execution
(Test Module definition)
3
5
(Module testing)
4
Implementation & Integration

Figure 4-4: V-model for the testing process

Besides the already mentioned requirement specification and test case conceptualization phase, the latter one has to be separated in further sub-modules, similar as done for the vehicle conceptualization phase. However, instead of the vehicle itself, the test environment is now in the focus of the V model and thus these sub-modules differ significantly.

A general pattern of a test environment is about UUT calibration as shown in Figure 4-5.

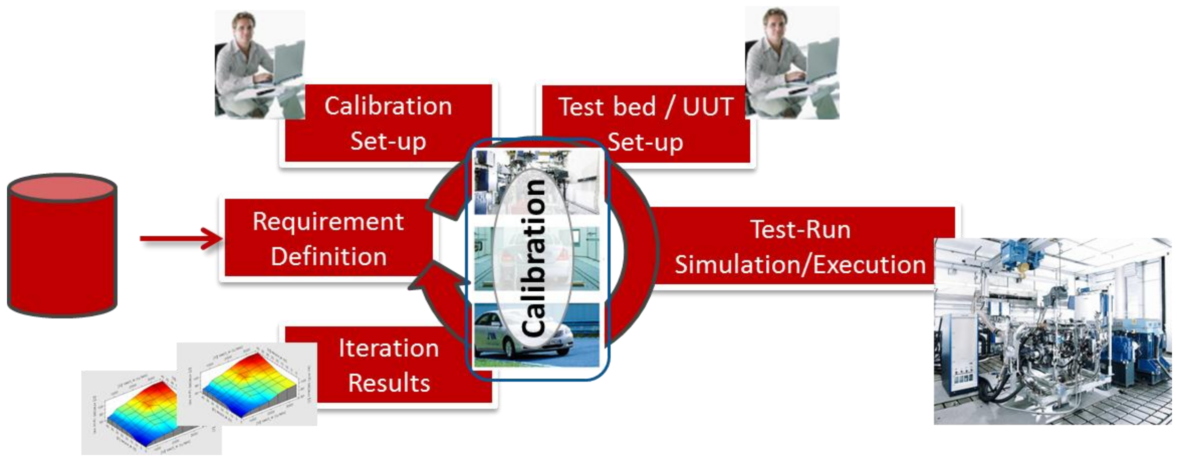


Figure 4-5 Test and calibration iteration pattern

The *test case conceptualization* is divided into two set-ups (*test modules*): the calibration and the test bed set-up. The test bed set-up configures the testing environment (with further sub-modules such as needed measurement devices, UUT configurations, etc.), while the calibration set-up is specialized on tuning selected parameters of the UUT in order to fulfil the given set of requirements. The specification of these set-ups with all there sub-modules finally leads to the implementation and integration of the overall test set-up, which can be executed with a set of given test-runs.

In many cases, the test modules *test case execution* and *calibration* are clearly separated: During a test-run execution, measurement results are linked to given test run input vectors. These pairs of data are used in corresponding calibration tools to create a calibration model that interpolates even not tested constellations and therefore support speeding up the calibration process as a whole.

4.2 Interoperability Challenges

In the previous sections, *the testing V-model* was separated from the classical *vehicle development V/W-model* order to change the focus of what should be achieved in both development processes. Since *testing a vehicle* remains a development element of the overall story of *building a vehicle*, the aspect of testing a vehicle automatically inherits the set of vehicle requirements. Or in other words, the set of *vehicle requirements* is extended by additional *testing requirements* such as demanded by the WLTP or a specific test bed environment as described in the previous section.

In this section, *interoperability challenges* for verifying both kinds of requirements under the special aspect of testing in different vehicle development phases (that correlates to the position in the V/W-model about *building a vehicle*) are described. These challenges are derived from the need that data within a (testing) V-model has to be interlinked properly to ensure traceability as described in the previous sections and are thus called *interoperability challenges*.

The key user story (as defined in the project's application document) of the UC 3.4.a is about full integrated *(model-based) requirement engineering* throughout the whole development process. This means that in case of *building a vehicle,* it has to be shown that a vehicle or parts of it (virtualized by simulation models or not) are fulfilling the requirements. This is only possible, if traceability is established between all artefacts that are needed to verify the requirements.

This is also true in case of *testing a vehicle*. All test artefacts that are needed to build up the *test case conceptualization* (as defined by the *testing V-model* in the previous section) have to be linked to the requirements and even have to be grouped themselves to a certain test case. Through its execution and the analysis of the testing results, finally, the requirements are verified (semi-)automatically.

According in the V/W-model about building a vehicle, the development phases are separated starting with the *requirement specification* and leading to concrete *implementation* of these requirements. The intermediate step of vehicle conceptualization could be subdivided in conceptualization of the main parts of a vehicle, namely the engine, powertrain and chassis conceptualization.

In the following, a systematic mapping of the *vehicle development phases* to the major *testing phases* is established. As stated in the previous chapter, each testing phase come up with their own challenges, tool chains, but need to be synchronized regarding their (testing) requirements, data sets and test cases.

4.2.1 Mapping the V/W-model to test phases associated testing V-models

Figure 4-6 illustrates a mapping of the W-model about building a car to test phases. These mapping and the resulting test phases are then described in more detail.

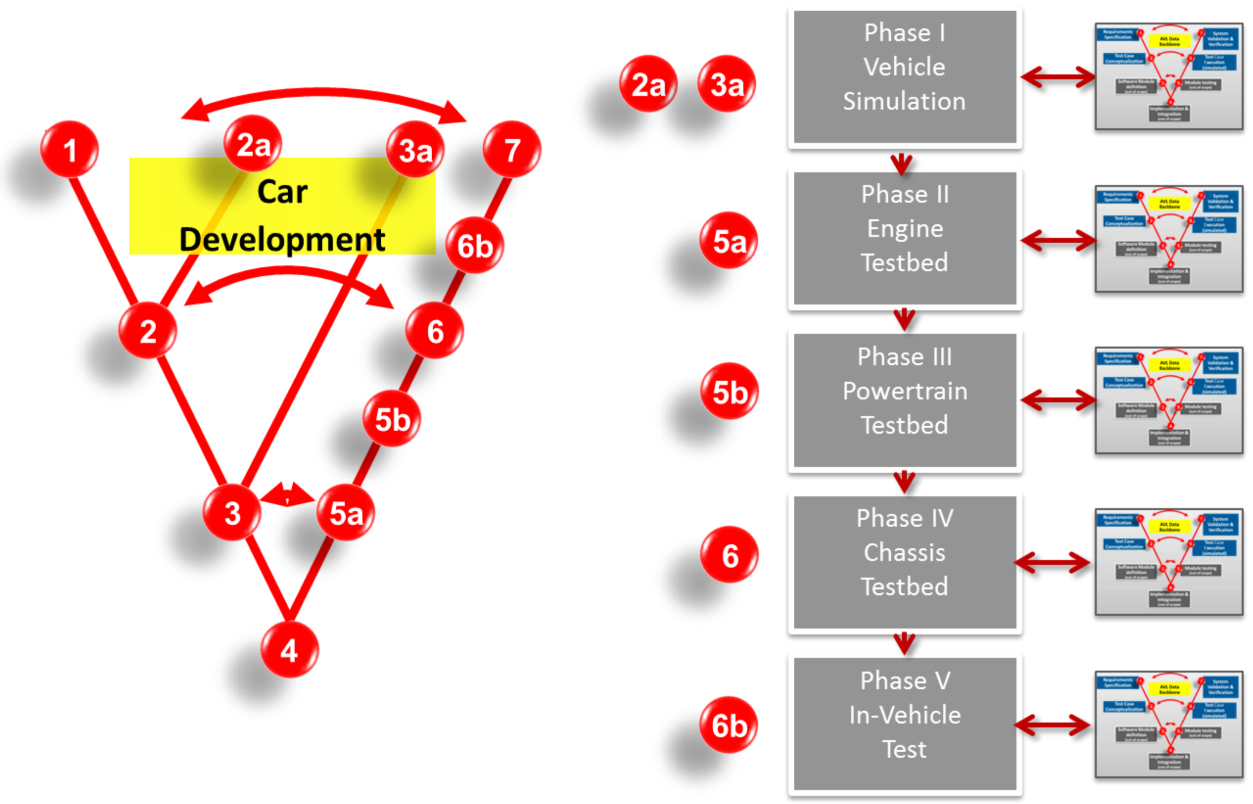


Figure 4-6: Mapping the W-model to vehicle test phases

Let's first describe the W-model about building a vehicle once more under the aspect of testing.

1. _Vehicle conceptualization (2):_ This phase can be accomplished by virtually modelling the vehicle according to its _requirement specification (1)_. These requirements can then be validated on vehicle level by _simulating the vehicle model for certain test cases (2a)_.
2. _Engine, powertrain, and chassis conceptualization (3):_ Simulation models are an essential aspect in the conceptualization of these three vehicle components. Associated simulation models are designed with a high detail level for each of these parts. The execution of these simulation models according to appropriate _test cases (3a)_ validates the results of the overall vehicle conceptualization.
3. After a concrete _implementation phase (4)_, the concrete sub parts of the vehicle need to be tested against its conceptualization specification. In our use case, this is done for the _engine (5a)_ and the _powertrain (5b)_ with appropriate _test beds_, whereas the powertrain is comprised already of that engine, which was tested at the previous testing phase (5a). In the _integration phase (6)_ the subparts are assembled and verified using an _integration test (6)_ that has to validate the overall _vehicle_

conceptualization (2). This is usually done by a chassis test bed[3], whereas the concrete environment is still simulated (this may also the vehicle driver).

4. Finally, the virtual driver and environment are replaced by real ones and tests are performed by so-called *in-vehicle tests (7)*, which finally validate the *requirements of (1)*.

Based on this description, testing phases can be assigned to specific development steps of the presented W-model, whereas each testing phase is described by its own testing V-model as illustrated in the picture above. In our use case, five different *testing phases* can be separated from the description above:

1. *Testing Phase I:* Performing vehicle simulation (associated with development phases *2a* + *3a*)
2. *Testing Phase II:* Applying an engine test bed (associated with development phase 5a)
3. *Testing Phase III:* Applying a powertrain test bed (that includes engine testing; associated with development phase 5b)
4. *Testing Phase IV:* Applying a chassis test bed (associated with development phase 6)
5. *Testing Phase V:* Performing an in-vehicle test (associated with development phases 6b)

These development phases are by no means complete (further phases such as HIL/SIL testing could be interlaced). However, additional phases do not abandon the principle concept of separate testing V-models linked to a specific vehicle development phase. In addition, however, this does not exclude a possible overlap of methods and tools across the five testing phases (for instance, the simulation of the environment takes place in any test scenario except the in-vehicle test).

The *interoperability challenge* is now the transfer of the created data set from one development phase to the subsequent. In the first stage of the CRYSTAL project, the focus will be set on *Phase I* and *Phase II* in order to understand and solve these *interoperability challenges* in more detail.

4.2.2 Data Categories and applied Tools in UC3.4a

In the Section 4.2.1, it was stated that every *testing V-model* is comprised of its own methods and tools. Usually, independently developed tools (if not integrated in a certain tool suite) have its own data sources and sinks. Tool interoperability is then often established by point-to-point integration, data transformation or data import features. This is also true in many times for AVL testing tools, which are traditionally developed as separate products and interoperability is made ad-hoc where needed..

A drawback of that is not the full potential of interoperability may be addressed. For instance, if two separate products of two separated *testing phases* (and thus V-models) are used by two different development teams, similar task that has to be done in both phases are performed redundantly and may be also slightly differently for no obvious objective reason. Due to that, data created during the first phase is not compatible for the second phase and so on.

Consequently, the first step towards interoperability is to categorize the data sources and then try to understand how the resulting *data categories* are instantiated in the several testing phases and how they are related across these phases as schematically sketched in Figure 4-7.

[3] A chassis test bed is equipped with complete vehicle put on some moving rolls, whereas the drive and the vehicle environment is still simulated.

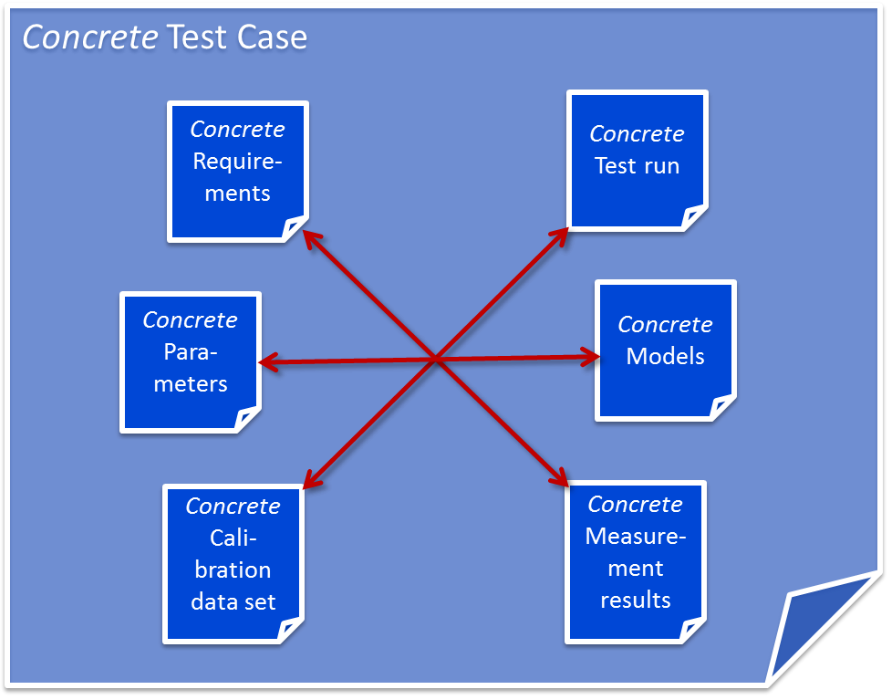


Figure 4-7: Test cases as data container for the various testing data artefacts

In addition, to the data categories relations, a kind of data container is needed that assigns concrete instances of the data categories to a concrete test case. Such a containers enables test configuration analysis to answer questions such as: Which calibration and testing set-up has been used to produce a specific set of measurement result?

In respect to the five *testing phases*, the following *data categories* are derived and its relation to the use case scenario (as defined in Section 3.1) is described:

- *Requirements* for vehicle development and testing: Requirements are represented in different forms. At the beginning of a project, they are formulated by informal textual representations. In order to verify requirements (semi-)automatically the need to be (semi-)formalized. (Semi-) formalized representations would be for instance boilerplate expressions.

  APPLYING THE *USE CASE SCENARIO*

  In our use case scenario, for instance, the WLTP specification is the foundation for the testing requirements (in addition to the vehicle requirements). Thus these specifications have to be formalized. We will examine three (semi-)formal representations: *Boilerplates*, *requirement models* in form of SysML models (e.g. state charts) and *sequenced based requirement* representations. All these representations are created by different tools and need to be interlinked to enable traceability. Examples for vehicle and (WLTP-based) testing requirements are listed in Section 4.1.2.

  The requirements should be represented in as much representations as possible/useful. The following tools are considered for the various representations:

  - HP Quality Center (currently used within some departments of AVL)
    - Stores an informal requirement representation
    - an ALM tool such as PTC Integrity (already used within some departments of AVL) is considered to be an alternative tool for this use case
  - ViF Boilerplate Prototype tool
  - Artisan Studio (for SysML representations)

- *Simulation models* play a central role in testing. In most testing phases - not only in phase I – simulation models are heavily used. If testing the powertrain (that include engine testing) the remaining vehicle has still to be virtualized (rest vehicle simulation). Even if a complete vehicle is tested on a chassis test bed, the environment (and may be also the driver) has to be simulated according the testing scenario.

APPLYING THE *USE CASE SCENARIO*

In our use case, we are focussing on the testing phases I and II (as described in Section 4.1.1). Usually in these phases the following type of models and tools are applied:

- *Vehicle models*: These models simulate the physical behaviour of either the whole vehicle or parts of it (such as a particular engine). Depending on the concrete application (needed details, real-time properties, etc.), either *Simulink* models or AVL *Cruise* models are used for that.
    - AVL *Cruise* is a simulation tool environment with certain specific simulation domains such as computational fluid dynamics (CFD) simulation, mechanical or system (i.e. vehicle) simulation.
    - *Mathworks Simulink* is a de-facto industry standard for a wide range of simulation applications. In our case, Simulink models are applied mostly for the simulation of control software.

- *Test runs* define the input data of a particular testing scenario either for simulation or for a particular physical unit under test (UUT), such as the engine, the powertrain or the whole vehicle. The WLTP, for instance, contains exact definitions, which events have to happen at what time during a test run. Events could be a certain throttle position or particular driving manoeuvres (breaking, steering, etc.)
- *Parameters*: To configure a certain testing procedure, a huge set of parameters is available, which can be assigned to different *parameter categories*. These *parameter categories* are used to configure *test beds*, the *test method* or the *UUT* itself or even its environment if simulated. This means parameterization of simulation models is subsumed here as well.

APPLYING THE *USE CASE SCENARIO*

In our use case, we try to establish a clear separation of these categories. Nevertheless, the current state is the parameters are spread over different tools in different representations (especially if different test phases are considered). The table below considers the assignment of two typical vehicle parameters to the corresponding tool and testing phase. In order to establish tool interoperability, it has to be considered how these different parameter representations can be mapped to each other within and across testing phases.

Parameter	Tool	Testing phase
Cylinder capacity	AVL Cruise	I
Cylinder capacity	AVL Puma	II, III
Cylinder capacity	AVL Cameo/Creta	I, II, III
Number of gears	AVL Cruise	I, III
Number of gears	AVL Puma	II, III
Number of gears	AVL Cameo/Creta	I, II, III

Table 3: Examples for parameters

- *AVL Cameo/Creta:* AVL Cameo performs the modelling and optimization tasks based on the measured engine response. AVL Cameo's primary use is for (automated) UUT calibration. AVL Creta is responsible for calibration data management.
- *AVL PUMA:* Integrates all system functions of a test bed and is used for testing tool and test bed component management.

- *Calibration* data are related to parameters but focus on special UUT parts such as the control software of an engine or a powertrain. In other words the calibration date does not configure the testing procedure, but fine-tunes the UUT according to the results of the testing procedure. What is subsumed as calibration data depends on the actual testing phase. Generally spoken, calibration data are concrete values of parameters that could be easily changed in a particular a testing/calibration iteration. If simulating a vehicle, more or less everything can be seen as calibrate able (e.g. even the cylinder capacity of a particular engine), whereas in later phase (e.g. on an engine test bed) this is not possible any more. In Section 8.4, the engineering method of calibration iterations are described in more detail.

APPLYING THE *USE CASE SCENARIO*

In the context of our use case scenario, the table below chooses four parameters as calibration candidates and assigns them to corresponding tools and testing phases. Note that the parameter 'Cylinder capacity' can be only assigned to testing phase **I**, since it is not possible to change this engine design parameter once the engine is physically build and tested in a corresponding testbed.

Parameter	Tool	Testing phase
Cylinder capacity	AVL Cruise M	I
Injection time	AVL Cruise M	I
Injection time	AVL Puma	II, III
Injection time	AVL Cameo/Creta	I, II, III
Shifting thresholds (for shift up)	AVL Cruise	I, II
Shifting thresholds (for shift up)	AVL Puma	III
Shifting thresholds (for shift up)	AVL Cameo/Creta	I, II, III
Shifting thresholds (for shift down)	AVL Cruise	I, II
Shifting thresholds (for shift down)	AVL Puma	III
Shifting thresholds (for shift down)	AVL Cameo/Creta	I, II, III

Table 4: Examples for parameters

- *Measurement results* is the data output of a concrete test-run execution. These results need to be analysed and requirements have to be validated against these analyses. Furthermore test-run inputs and measurement outputs are building the basis for the definition of a calibration model.

APPLYING THE *USE CASE SCENARIO*

The following requirements could be associated with concrete measurement results. The exact content and meaning of measurement results often differs from case to case and especially between different testing phases. Consequently a mapping and unique interpretation of the measurement results is considered as an interoperability challenge. The table below associates selected vehicle and testing requirements to particular tools that manages them in various testing phases. If a 1:1 mapping of measurement result and the related requirement is not possible measurement result post-processing is provided by separate tools (e.g. AVL Magic or Concerto)

Related requirement	Tool	Testing phase
Emission limit requirement	AVL Cruise	I
Emission limit requirement	AVL Santorin	II, III
Emission limit requirement	AVL Cameo	I, II, III
Maximal speed	AVL Cruise	I
Maximal speed	AVL Santorin	II, III
Maximal speed	AVL Cameo	I, II, III
Time for acceleration 0 to 50 km/h	AVL Cruise	I
Time for acceleration 0 to 50 km/h	AVL Santorin	II, III
Time for acceleration 0 to 50 km/h	AVL Cameo	I, II, III

WLTP_RQ1	AVL Cruise	I
WLTP_RQ1	AVL Santorin	II, III
WLTP_RQ1	AVL Cameo	I, II, III
WLTP_RQ2	AVL Cruise	I
WLTP_RQ2	AVL Santorin	II, III
WLTP_RQ2	AVL Cameo	I, II, III

Table 5: Examples for requirements

AVL Santorin: Based on ASAM-ODS standards, the AVL Santorin Server provides the infrastructure for consolidated storage and manufacturer independent analysis of measurement data.

4.2.3 AVL Data Backbone

In the previous sections of this chapter, development and testing processes, data categories and their assignment to tools are explained. Various tools are used to create different kinds of artefacts throughout the testing process. Most of them are currently stored locally or in a non-transparent manner (and thus are hardly to access), which makes data-reuse and traceability complicated or even impossible.

The AVL Data Backbone is a generic concept to overcome these limitations. Figure 4-8 below illustrates the basic idea of this concept.

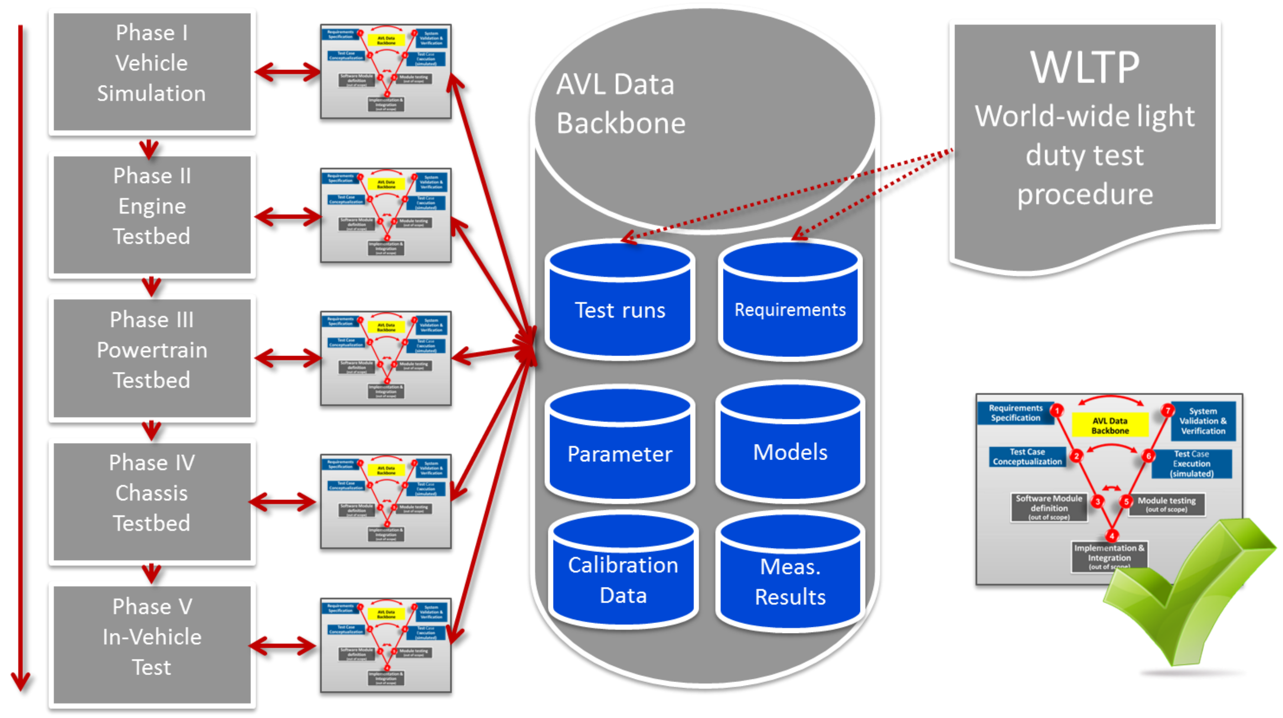

Phase I Vehicle Simulation
Phase II Engine Testbed
Phase III Powertrain Testbed
Phase IV Chassis Testbed
Phase V In-Vehicle Test
AVL Data Backbone
Test runs
Requirements
Parameter
Models
Calibration Data
Meas. Results
WLTP
World-wide light duty test procedure
Requirements Specification
AVL Data Backbone
System Validation & Verification
Test Case Conceptualization
Test Case Execution (simulated)
Software Module definition (out of scope)
Module testing (out of scope)
Implementation & Integration (out of scope)

Figure 4-8 The AVL Data Backbone concept

The AVL Data Backbone a kind of single-source-of-truth for all tools and data categories applied in all testing phases represented by different testing V-models (and thus different tools). With this concept consistency among the development processes should be enabled and effective frontloading of development tasks becomes possible.

It is important to understand what consistency means in this context. For some situations it is maybe sufficient that a central data repository ensures the single-source-of-truth concept, i.e. a unique storage location and transparent way of data access for the involved applications. In practise, however, a single-source-of-truth concept does not necessarily causes consistency of data content (e.g. re-using the same set of calibration data throughout two testing phases). It may happen that for various reasons (e.g. if different naming conventions are common in different development phases) two variants of calibration data sets are created at two testing phases.

A stronger meaning of consistency may forces that the data content of the several data categories is aligned with the different testing phases. For instance, with a fully consistent data set, the parameters, calibration data, and measurement results of two test-runs in different test phases phase became directly re-useable and comparable. It is part of the project to evaluate, which interpretation of consistency is sufficient to overcome the most important limitations of today's systems.

Figure 4-9 illustrates a possible concept based on the OSLC concepts. The various data categories are stored in one or even more (3rd party) data providers. The data consists of all the details created by the related authoring tool and only these authoring tools are able to fully interpret and modify this kind of data. OSLC adapters, however, abstract from these details and provide only a reduced data model per data category (also called OSLC domains). These (potentially standardized) OSLC domains are designed in a minimal manner in order to just fulfil the needs for defining data interrelations and navigation across the data categories. On top of this minimal OSLC data structure a *uniform workbench* could navigate over this data structure, without the need of understanding all the details the authoring tools have to deal with. If a deeper data analysis or modification is needed, the workbench just delegates this task by invoking the corresponding tool with the appropriate OSLC link or requests an appropriate data artefact representation.

In addition, an AVL customer has the freedom of choice which kind of data backbone he wants to use. For instance, a classical ALM tool such as PTC Integrity may provide important features such as variant and version management. However, this feature may not be needed by every customer, consequently another data provide (e.g. an AVL data backbone) is sufficient. With the use of OSLC both the authoring tools as well as the uniform workbench does not depend on which data provider is in use.

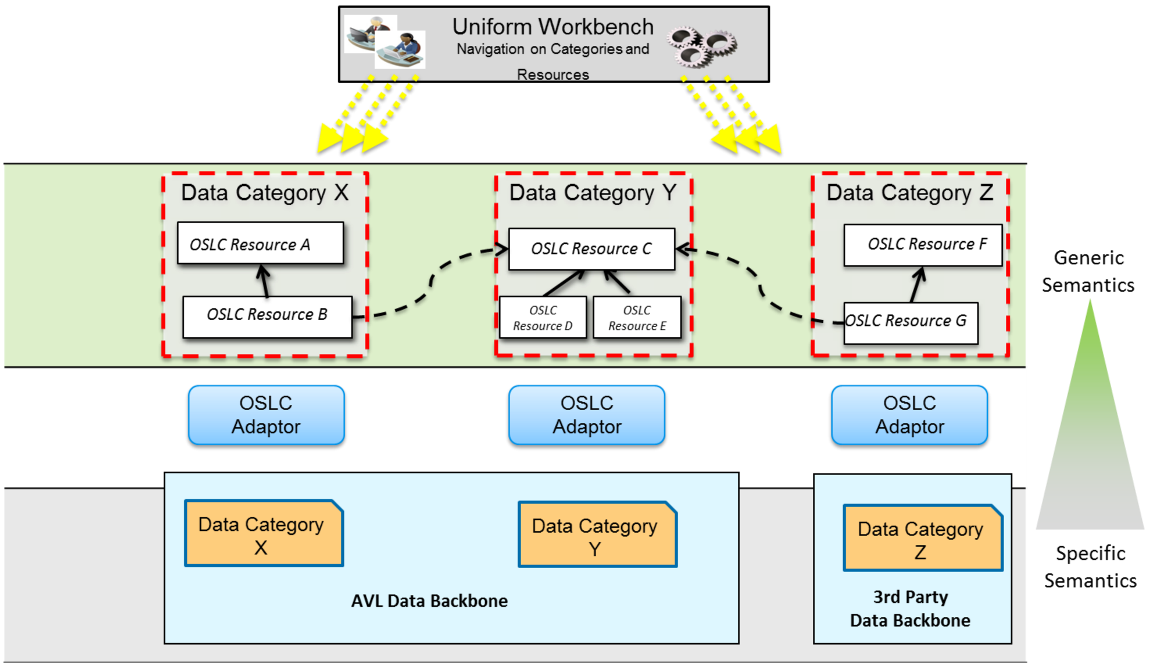


Figure 4-9 Integration of OSLC and the AVL Data Backbone concept

In terms of OSLC, it is mandatory to develop a corresponding OSLC model that interlinks the data categories appropriately. One purpose of the CRYSTAL project is to develop and standardise such appropriate OSLC resource models by analysing the interoperability challenges of the provided use cases.

Applying to the use case scenario

In the context of the use case scenario (as defined in Section 3.1), the data backbone plays a central role. Starting from the (formalized) requirements, all the data artefacts created in the following testing phases should be stored and managed by such a data backbone concept. The selected tools of choice will be *AVL Santorin* and *AVL Navigator*. AVL Santorin is traditionally used for managing measurement result data and complies with the ASAM ODS standard. On the other hand, it is also a database that is able to hold other data as well. In terms of version and variant management, however, the possibility is limited and alternative/parallel solutions may be considered for the use case scenario (e.g. by integration of a PLM tool). The AVL Navigator is able to navigate on top level elements of data categories, but it delegates functionality of editing and interpreting the data content to specialized tools. Thus the AVL navigator naturally fits into the data backbone concept sketched above and would be an instance of a uniform workbench.

Based on the use case scenario description, these tools should be applied to demonstrate a fully integrated requirements engineering process including requirement verification and data migration over development phases. In terms of OSLC, is mandatory to develop a corresponding OSLC model that interlinks the data categories appropriately. The description of the engineering methods in Chapter 8 delivers a first draft of such a data model, which need to be evaluated and adapted by applying the use case scenario during the CRYSTAL project.

5 Use Case Process Description of UC 3.4b

UC3.4b *use case process* is based on the following three different *use case scenarios* in order to improve the software component development at AVL-R:

1) Model Exchange and integration of a System-of-systems (SoS) platform.
2) Establishing an integrated tool environment for embedded controls development.
3) Improving of the software component development via an efficient variant handling within V-model.

Every use case scenario has to deal with different interoperability challenges, which are described by the following sections.

All Use Case Scenarios base on the same V-model with described activities in Chapter 6.1.

5.1 Use Case Scenario 1: Model Exchange and Integration of a System-of-systems (SoS) platform

Due to a model-based software development process, AVL-R provides appropriate simulation models of vehicle control systems. To improve the quality of these control system models, proper simulation models of the plant and the physical vehicle behaviour enables development frontloading. For vehicle simulation, AVL-R is using the simulation environment AVL BOOST RT[4].In addition the imported AVL BOOST RT have to be integrated and configured in the versatile System-of-systems (SoS) platform where different hardware or software systems to create a new execution environment.

5.1.1 Enabling simulation and plant models exchange

BOOST RT is a real-time capable, system-level, engine simulation tool dedicated for the investigation of transient operating conditions offline in desktop applications and online in HiL environments. It is the consequent next step to further integrate the well-established BOOST 1D Cycle Simulation tool and CRUISE, the vehicle dynamics, fuel consumption and emission simulation package. BOOST RT focuses on three main application areas used in the different engine development phases:

a) **Concept Development Phase**: BOOST RT supports a fast setup and simulation of engine design variants without considering all details of pipe gas dynamics to get guidelines on engine performance.
b) **Powertrain Design Phase**: BOOST RT engine runs drive cycles fully coupled with a detailed vehicle model (built in CRUISE) resolving both engine and vehicle dynamics with adequate level of modelling depth within reasonable computational times.
c) **Component, Powertrain, Vehicle Test Phase**: BOOST RT engine runs as plant model to support the development, calibration and testing of engine control functions.

The exchange of the configured engine model is currently done via time-consuming personal request to the responsible persons and should thus be improved by a tool supported approach.

Tool supported approach means that Integrity is able to have a link to other data providers such as the AVL data backbone (developed by UC3.4a and described in Section 4.2.3), which shall be realized via an OSLC adapter on Integrity side.

The *interoperability challenge* for this use case scenario is thus the model exchange between a *AVL data backbone* and PTC Integrity tool.

[4] Based on AVL PRODUCT DESCRIPTION BOOST RT

A proposal for such a common simulation model data backbone is given in Section 4.2.3. As described in more detail in Chapter 7, this data backbone relates the two sub use cases UC3.4a and UC3.4b.

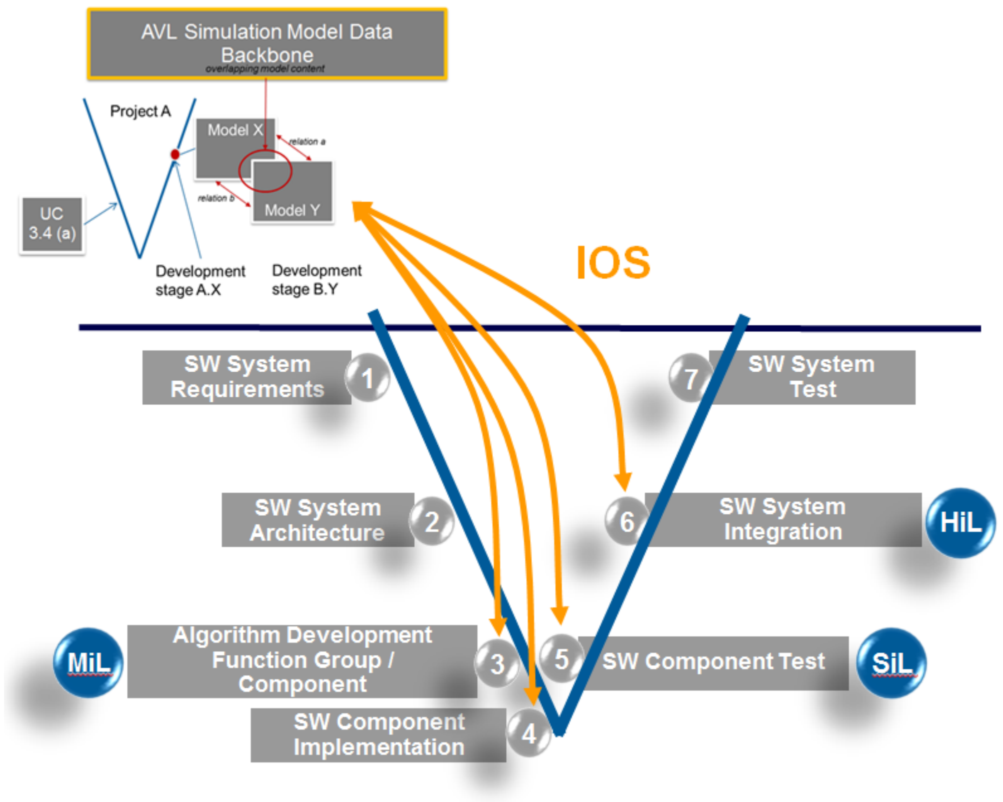


Figure 5-1: Overview Use case Scenario 1

Figure 5-1 shows the interoperability challenge between *Data Backbone* (upper part) and the different development steps of the *V-Cycle* in INTEGRITY (bottom part).
The improvement is given by a simulation model exchange (i.e. BOOST RT models) between the *AVL data backbone and the AVL-R tool environment based on PTC Integrity.* An IOS-based interface between the involved tools ensures improved tool interoperability compared to the current situation.
The concrete scenario simulation model exchange has to be evolved during the CRYSTAL project (e.g. importing simulation models vs. linked-data approach.

5.1.2 Integration and configuration of a versatile System-of-System (SoS) platform

In UC3.4 b the exchange models will be additionally used for the design and development of a flexible and configurable System-of-Systems (SoS) platform including the necessary tools for configuration of hardware, software, application, and communication aspects. The motivation for this step is that many upcoming applications in the automotive domain require versatile and adaptable systems (or system platforms) that can be tailored to specific application area requirements and constraints. To reach this ambitious goal (*interoperability challenge*), research and development not only on a platform level but also in the area of engineering processes and tooling needs to be undertaken. In particular, ways to integrate different subsystems and applications to a common platform and to provide tailored configuration tool-chains that support this process are the main work items of this use case scenario.

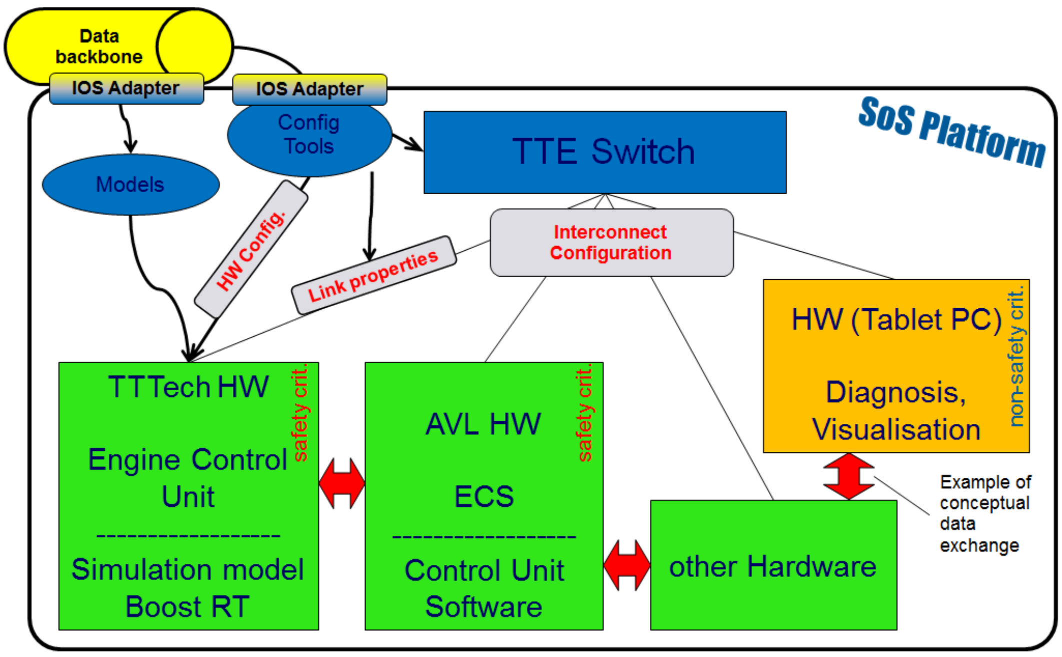


Figure 5-2: Integration of a configurable SoS Platform with Use Case requirements

Figure 5-2 illustrates the user scenario targeted in this use case. Main aspects are the interconnection of multiple hardware platforms towards one system-of-systems platform and their variable interconnection. The configuration of communication and platform aspects will allow to instantiate a tailored system platform for specific use cases out of more generic blueprints. In this use case scenario, the usage of a simulation model – imported from the *Simulation Data Model Backbone* – within an embedded verification platform will build the basis of the SoS Platform. Interfacing to the IOS, configuration tools can obtain requirements from there and could even gather information from simulated system parts (*Simulation Data Model Backbone)* integrating all these in the SoS Platform. In addition, the requirement sets get the possibility to be instanced variably and variant management aspects can be supported. A further topic is the practical test and validation of the use case and its applications.

5.2 Use Case Scenario 2: Integrated Tool Environment for embedded controls development

The interconnection between several tools and their interoperability capabilities and the integrated tool environment (AVLab) is currently realized in a not standardized way and has to be improved.
For an improvement it has to be analysed about an IOS implementation. Furthermore, the “Integrated Tool Environment” provides a single point of control during all development steps including MiL/SiL Tests and should be extended to support HiL / Engine Bench tests which are currently done manually.

The *interoperability challenges* are located in the harmonization improvement between the different tools and there interfaces. The aim is to provide a standard interface for the “Integrated Tool Environment” in order to harmonize interfaces, facilitate the substitution of tools, and to be more independent of concrete tool versions. Furthermore, seamless traceability between all artefacts should be supported by adopting IOS concepts.

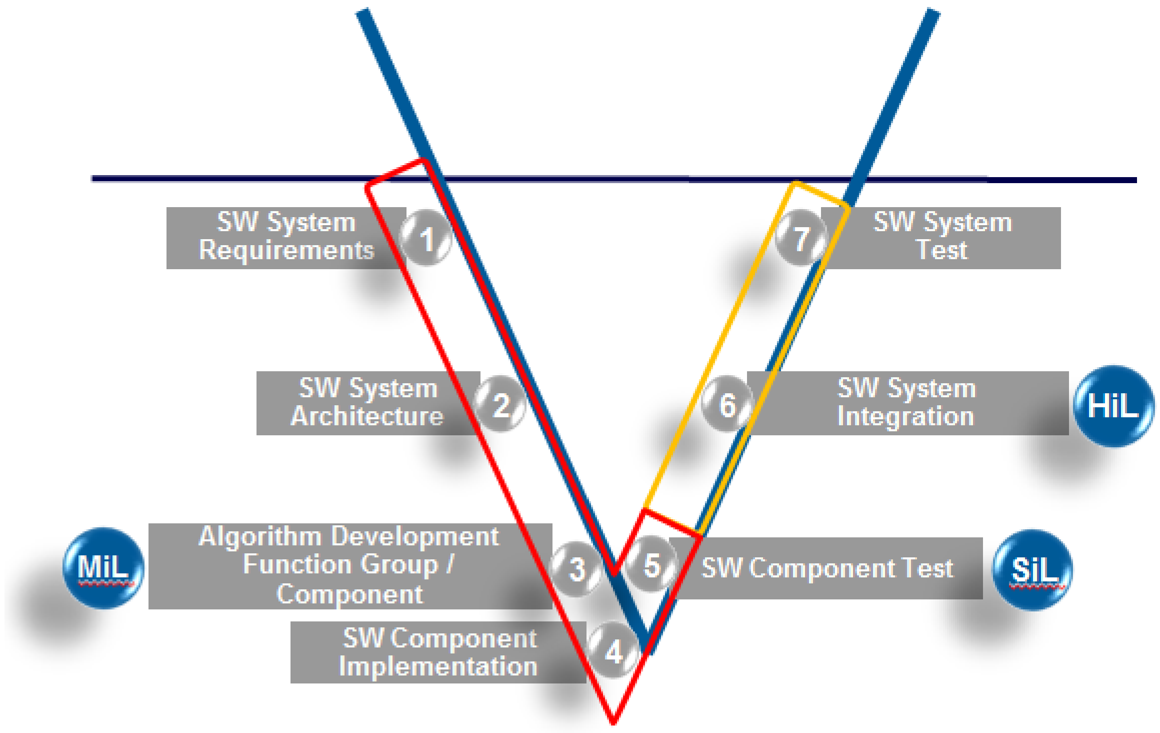


Figure 5-3: Overview Use Case Scenario 2

Figure 5-3 shows the interoperability challenge within the V-Cycle of AVL-R. The red highlighted part shows the current implementation of AVL-R's integrated tool environment which have to be improved by adopting IOS concepts. For information exchange between the development steps within the yellow highlighted part must be implemented and simultaneously enhanced by applying IOS concepts as well.

Overview about the current tool landscape:
Various tools are used for the different development steps throughout the software development V-model as illustrated in Figure 5-4. Additionally is shown, that AVLab currently supports the V-model starting with requirements until the SiL-test and do not support the HiL and Vehicle tests.

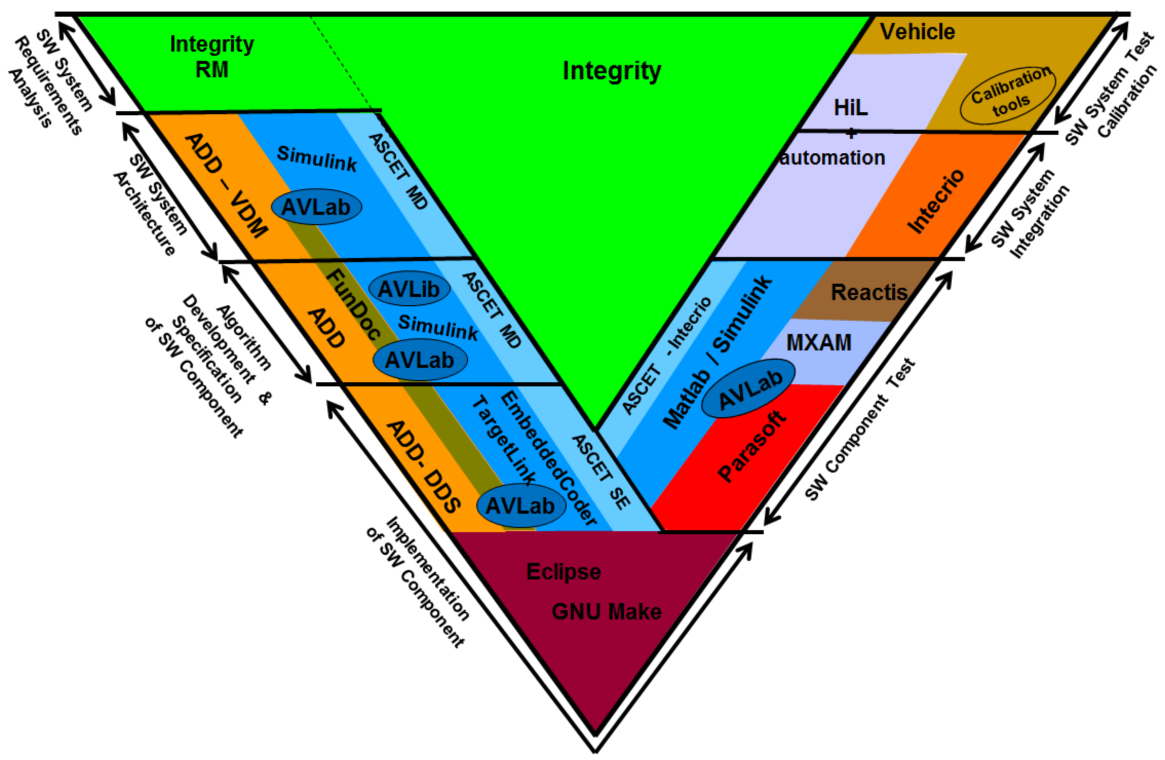


Figure 5-4: Used Tools in AVL-PTE Controls development process (overview)

AVL-R is currently using the *Integrated Tool Environment* (AVLab) as a common user interface, which is a single point of contact for all related tools, i.e. AVLab bundles several tools supporting each development activity steps from modelling over testing to code generation, which should lead to a **seamless tool chain** as listed in the following:

- ADD (Visu-IT!) ⇔ Simulink: Data synchronization via SyncTool
- Integrity (PTC) ⇔ Matlab: Support and Integration of PTC Integrity Source in AVLab
- Simulink (The Mathworks) ⇔ AVL Concerto: allowed Concerto Plots for data visualization via AVLab
- MXAM (MES) with AVL modelling rules is started from AVLab
- MIL / SIL / back2back tests supported by AVLab
- Code Generation

It also provides some kind of guidance for the developer through the development process (requirement management, architecture, model development, tests, and code generation).
Further background is the harmonization of the process & tool environment for PTE Controls.

Key points for this harmonization are:

- Component based development approach is enforced.
- **Scheduling** of components in function groups (and domain) is enforced by model template in AVLab.
- **SW Architecture** is enforced by ADD as architecture tooling.
- **Code generation** for SW components is supported by tool environment in Embedded Coder and Target Link. Code generator configuration is unified.
- **Build environment** (= generation of flash able hex file) is based on generated and archived C-Code.
- Integrated **test framework** enforces common way of testing.

AVLab currently supports function development from model development over model testing to **code generation** in a Matlab/Simulink environment, with Embedded Coder or TargetLink as code generator. In addition, AVLab shall support the **methods** in engineering area:

- **A three Level Architecture** is the basement of the component-based development
- **Naming** is ensured by the usage of the Name Checker inside ADD
- **Modelling Guidelines** are checked by the usage of MXAM (model style checker)
- **Product Documentation** is ensured by the usage of FunDoc (Visu-IT)
- **Verification/Validation** is supported via MiL, SiL and Back-to-back testing in AVLab
- **Coding Guidelines** are supported via Code Generation Helpers (Embedded Coder Toolbar or TL Code Generator)
- **Build** is directly supporting component-based approach

The current implementation of AVLab increases the efficiency / quality of function development.

- Provide a template for modelling, with operating system to allow a simulation closer to the reality than a pure Simulink simulation.
- Simulink toolbar, with shortcuts for faster action in Simulink
- Traceability between Model, ADD Container, Integrity

5.3 Use Case Scenario 3: Improvement of the Development via „Efficient Variant Handling within V-Model

AVL-R provides software solutions for various domains (e.g. automotive, shipping, trucks). Solutions for single domains can be very different, but within one domain there is often a huge potential for reuse.
Currently, there is no explicit and systematic variant handling at AVL-R. SW variants are stored in PTC Integrity, without mechanisms to search for or select a specific variant.
Furthermore, there is no detailed definition of variant handling and usage throughout the development process.
Variant handling cannot be considered as an *interoperability challenge* on its own, but has to be considered for the IOS specification in general. For instance, if a linked-data approach such as OSLC is applied, a link management concept, which can deal with variants of the associated data artefacts, has to be provided.

Figure 5-5 shows the needs of variant handling within the V-Cycle of AVL-R.

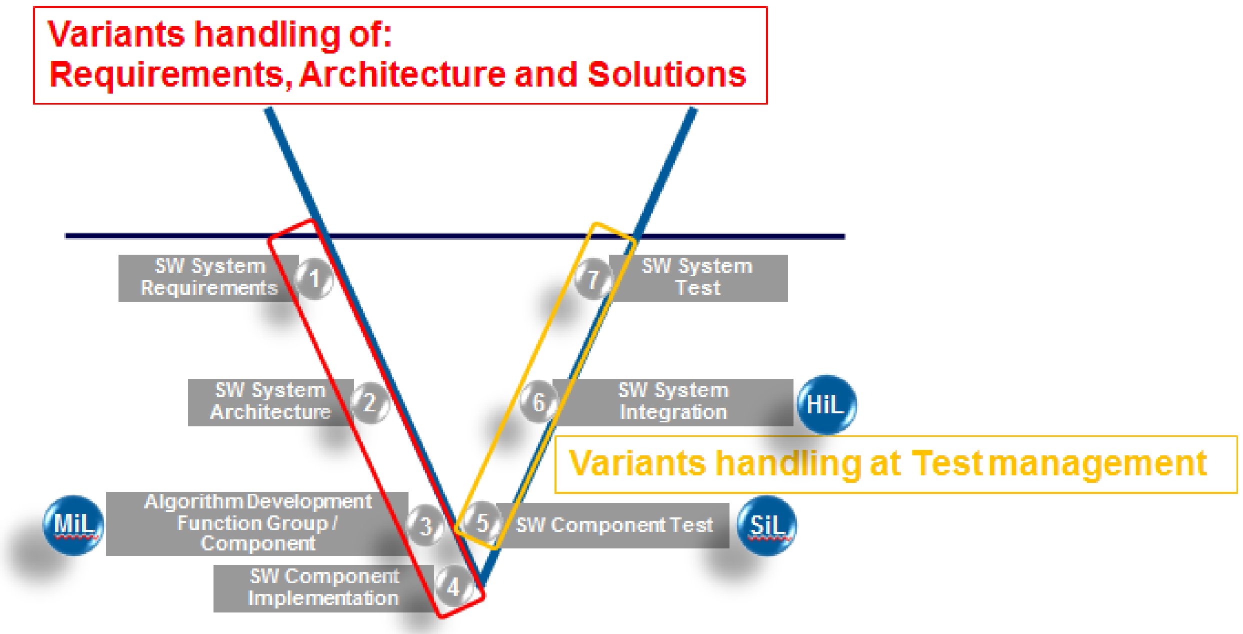

Variants handling of:
Requirements, Architecture and Solutions
SW System Requirements
1
SW System Architecture
2
Algorithm Development Function Group / Component
3
MiL
SW Component Implementation
4
5
SW Component Test
SiL
6
SW System Integration
HiL
7
SW System Test
Variants handling at Test management

Figure 5-5: Overview User Scenario 3

The red highlighted part illustrates the part of the V-Cycle which is supported by INTEGRITY and used since a certain time. The yellow part is also supported by INTEGRITY but with less experience about the current implementation.

To improve the current existing implementation, the used variant handling has to be checked about usability. An improvement could be the implementation of mechanisms within INTEGRITY to get a suitable variant management. Therefore the current development steps must be analysed and a variant management for the complete V-model has to be implement in the ALM Tool.

In the following chapters, the several development steps are described the current implementation.

5.3.1 Basic definition

In order to set up a suitable variant management strategy, the following requests / definitions must be clarified.

- What is the definition of a Variant?
  Current AVL understanding of a variant:
  A variant is the developed solution requested from the customer.
- What is the content of a reusable Variant?
  Is it only a SW component? Does it also have linked test cases, requirements, and so on?
- Definition for an Efficient reuse of requirements, architecture, functions, SW, tests, tasks…
  … via new creation, copy, delta…
- Is it possible to build a common (configurable) code base or should there be a modularized structure of SW components?

Variant management is a task which has to be described throughout the development process. It has to be decided if there should be a centralized variant handling which consistently controls variability in all artefacts or if a separate approach should be used for each single artefact in the development process. The former seems to be more likely, but the concrete variant handling strategy should be developed in the course of this project. Below, we describe the needs for each development stage at AVL-R.

5.3.2 Requirements Engineering

Currently, there is no variant management for the requirements applied in the AVL-R software development process. In addition, there is also no traceability approach from the requirements to the implementation and the test management, so far. One main reason is the fact that requirements from customers have various formats and are often not machine process able and have to be entered manually.
Without linking requirements to SW components and corresponding test cases, there is no real benefit for developers to write proper requirements. It further prevents the automatic validation of the system based on the requirements and traceability. Including variant management already in the requirements phase could also help to locate existing SW components which have already been implemented by just filtering requirements.

5.3.3 Architecture

AVL-R is currently using the ADD tool (Automotive Data Dictionary tool) as an architecture tool during development. The tool already provides variant management functionality which is currently not used, because the current process does not support this tool feature. For that reason, it has to be decided if this functionality should be activated or a other approach will be used for the variant management.,
To handle the different variants / versions is needed to reproduce a former state of architecture.

5.3.4 Powertrain Software Framework

The Powertrain Software Framework (PSF) is an existing solution at AVL-R for software component reuse.
It is a method to classify software components in such a way, that a Powertrain Software Architecture can be built up in an easy way. Moreover it is a clip inside the configuration management system between the customer’s projects information, software system requirements and the representing solutions. All elements in the Powertrain Software Library are classified within the Powertrain Software Framework which base on a *Three Level Concept.*
The AVL Powertrain Software Architecture defines three levels which help to organize a software system:

Function Domain (PSF-Level1): is the highest level of software split for solutions inside the software system. The Function Domain is a physical component based split.

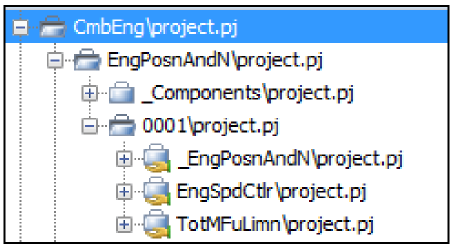
Function Group (PSF-Level2): is a sub element of the Function Domain containing a group of smallest elements, the function Component and their connectors inside. These elements can be shared or reused. All Function Components e.g. application, actuator and sensor software components for one physical component belong together.

Function Component (PSF-Level3): is the smallest element of the PSF Architecture. This is the lowest level where development files must be stored and handled. These elements can be shared or reused via a Function Group.

The main purpose of the PSF is the improvement of the current software development process in efficiency. One main aspect is the possibility to easily access information about available implementations and their requirements during a quotation, project start-up and during the development.
The usage of PSF supports …

- the development of one function / software only once (avoid double work on solution level)
- the reuse of functions in different applications
- the information exchange via usage of knowledge database
- the software systems built-up for different applications
- the reuse or development over different locations
- the development time reduction and efficiency increase by reuse
- the strategic decisions for future R&D projects

The following figure illustrates the current structure to store the Software components in Integrity. In this example is shown the Function Group Combustion Engine (CmbEng) which consists out of Engine Speed Control (EngSpdCtlr) and Tot Mass Fuel Limitation (TotMFuLimn) and composed within the Variant 0001. This Variant build’s the bases for an usage within a project.



```
⊟ CmbEng\project.pj
   ⊟ EngPosnAndN\project.pj
      ⊞ _Components\project.pj
      ⊟ 0001\project.pj
         ⊞ _EngPosnAndN\project.pj
         ⊞ EngSpdCtlr\project.pj
         ⊞ TotMFuLimn\project.pj
```

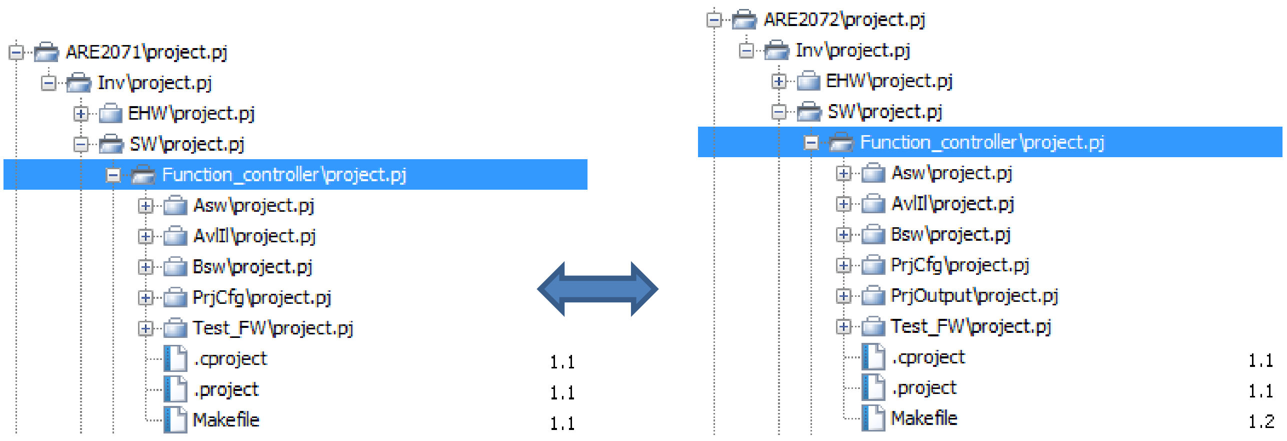
Figure 5-6: Variant storage within PSF-Library

Currently, it is only possible to provide a “one line free text” variant description, which is not sufficient in practice. There is no structured description of variants, e.g. using keywords. Therefore, the selection of variants has to be done manually.

5.3.5 Project

The implemented solutions are always based on customer requests. Until now, customers requested software solutions which have then been embedded in their own xCU’s. Nowadays, more and more requests about “multi project line” development must be supported.
A “Multi project line” is a project with a parallel customer request for more than on product (e.g. He requests in parallel a solution for a 4 Cylinder and 3 Cylinder engine) and a requested high overlap in functionality with and possible the reuse between these projects.

Because of no defined rules for this kind of reuse, each project handles the customer requests to the best knowledge without a dedicated way.



```
⊟ ARE2071\project.pj
   ⊟ Inv\project.pj
      ⊞ EHW\project.pj
      ⊟ SW\project.pj
         ⊟ Function_controller\project.pj
            ⊞ Asw\project.pj
            ⊞ AvlIl\project.pj
            ⊞ Bsw\project.pj
            ⊞ PrjCfg\project.pj
            ⊞ Test_FW\project.pj
              .cproject                      1.1
              .project                       1.1
              Makefile                       1.1
```

```
⊟ ARE2072\project.pj
   ⊟ Inv\project.pj
      ⊞ EHW\project.pj
      ⊟ SW\project.pj
         ⊟ Function_controller\project.pj
            ⊞ Asw\project.pj
            ⊞ AvlIl\project.pj
            ⊞ Bsw\project.pj
            ⊞ PrjCfg\project.pj
            ⊞ PrjOutput\project.pj
            ⊞ Test_FW\project.pj
              .cproject                      1.1
              .project                       1.1
              Makefile                       1.2
```

Figure 5-7: Variant Handling via separated projects (ARE2071 / ARE2072)

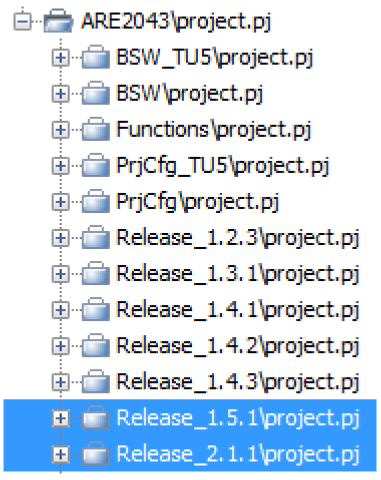

ARE2043\project.pj
BSW_TU5\project.pj
BSW\project.pj
Functions\project.pj
PrjCfg_TU5\project.pj
PrjCfg\project.pj
Release_1.2.3\project.pj
Release_1.3.1\project.pj
Release_1.4.1\project.pj
Release_1.4.2\project.pj
Release_1.4.3\project.pj
Release_1.5.1\project.pj
Release_2.1.1\project.pj

Figure 5-8: Variant Handling within one project line (Prj1: Release 1.5.1 / Prj2: Release 2.1.1)

Basically to find a suitable development process, a definition of a project must be done. This should be in relation to AVL's “MULTI PROJECT MANAGEMENT TOOL” (ProCalc).
It has to be pointed out, that there are currently two different kinds of projects at AVL-R:

- One R&D per project line. (see Figure 5-7)
- One R&D for both project lines (see Figure 5-8)

The issue of efficient variant handling arises if project A is the lead project and project B is the follower which reuse project A (100% reuse) or reuse project A with some own adaptation in the functionality.

The question about an efficient handling come up, e.g. if Project A is the lead project and Project B is the follower which only reuses components from Project A.
In this case following requirement derives:

- It must be possible to be efficient in project parallel development.
- It must be possible to change simply the lead from one project to another and vice versa.
- Assumption: the both projects are in one development “thread”.
- Definition of the most efficient handling for project variants.
- Definition of an efficient handling / reuse of all project items (e.g. also Tasks).

Furthermore we need concepts about the efficient creation, handling of project variants, effective reuse within a further project and its bug tracking at a reuse other project lines.
Therefore have to be defined:

- the way of project variant usage
  … when / how created (fixing at each Project Release)?
- the relation between a variant and a project release.
- the usage of the possibility to share (i.e. link) between projects.
- the possibility to calibrate SW-code.
  … currently not mandatory, but could be helpful.
- The influence about generic development, because this is currently not supported.

6 Detailed Description of the Use Case Process of UC3.4b

In addition to Chapter 2.2, this Chapter illustrates the detailed description of the UC3.4b *use case process* with its activities and the stakeholders and roles.

6.1 Activities

The applied V-model is comprised of various activities as illustrated in Figure 6-1. These activities are described in the following sections in more detail or implemented in the Excel sheet of Chapter 11.1. The operability challenges between the several activities are described in Chapter 9.

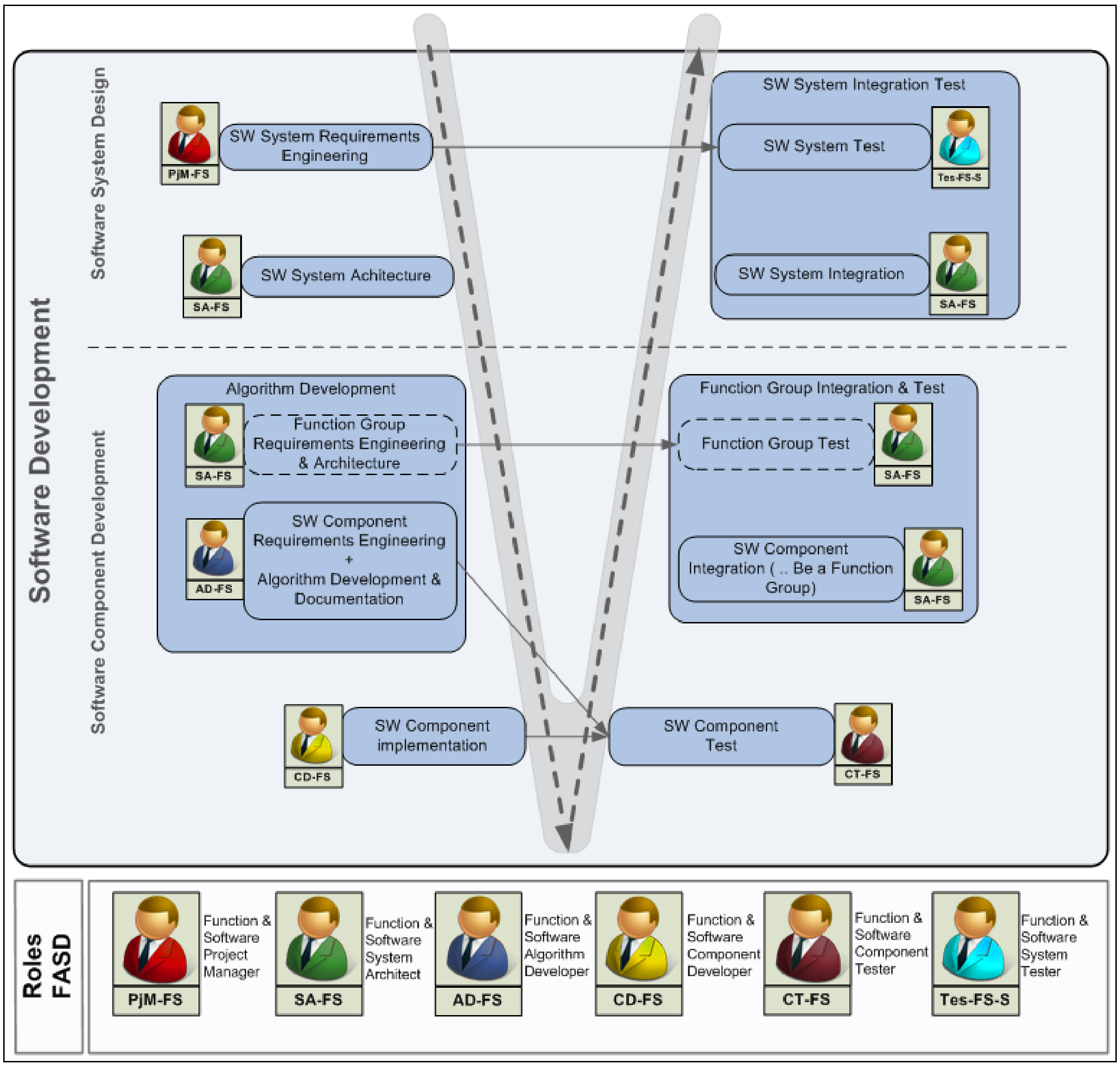

Software Development
Software System Design
Software Component Development
PjM-FS
SW System Requirements Engineering
SA-FS
SW System Achitecture
SW System Integration Test
SW System Test
Tes-FS-S
SW System Integration
SA-FS
Algorithm Development
SA-FS
Function Group Requirements Engineering & Architecture
AD-FS
SW Component Requirements Engineering + Algorithm Development & Documentation
Function Group Integration & Test
Function Group Test
SA-FS
SW Component Integration (.. Be a Function Group)
SA-FS
CD-FS
SW Component implementation
SW Component Test
CT-FS
Roles FASD
PjM-FS
Function & Software Project Manager
SA-FS
Function & Software System Architect
AD-FS
Function & Software Algorithm Developer
CD-FS
Function & Software Component Developer
CT-FS
Function & Software Component Tester
Tes-FS-S
Function & Software System Tester

Figure 6-1: AVL-PTE Development Process

The Activities of the Use Case process are:

Software System Requirements Engineering:

The SW system requirements that are delivered by the customer are analysed and put into a form that can be used as basis for the software development. Based on the SW system requirements, the SW system integration tests are specified.

Software System Architecture

The *software system architecture* is being defined, based on the *SW system requirements*. The *SW system requirements* are assigned to the SW components and handed down for further analysis to the function and software algorithm developers.

Algorithm Development

Algorithm development and the specification of the detailed SW component requirements are conducted here. The step already involves the set-up of a SW component model and the SW component documentation as well. Based on the SW component requirements, test cases for the SW component are developed. In addition to this, the SW components will be will grouped to so called Function Groups and get the similar tests scenarios as SW components.

Software Component Implementation

The SW requirements are implemented in the SW component, the model and documentation is refined.

Software Component Test

The software component is tested according to the defined test cases. The test report is delivered as well as the SW component including the documentation.

Function Group Integration & Test

The various software components are integrated and tested in the composition of a Function Group.

Software System Integration Test

The software system is integrated and tested according to the defined test cases. Test results and the SW system are delivered back to the customer

6.2 Stakeholders & Roles

Stakeholders	Role
Function & Software Project Manager	Definition of Software System Requirements
Function & Software System Architect	Definition of the Software System Architecture Creation of Software Component Test Report Creation of Test Software Component + Documentation Creation of Integrated SW System + Documentation
Function & Software Algorithm Developer	Definition of Software Component Requirements Creation of Software Component Model Creation of Software Component Documentation
Function & Software Component Developer	Creation of Software Component Documentation Creation of Software Component Model Creation of Software Component Code
Function & Software Component Tester	Test of Software Component
Function & Software System Tester	Creation of Software System Test Report Creation of Tested Software Package + Documentation

Table 6: Stakeholders & Roles

7 Detailed Description of Relation between UC3.4a and UC3.4b

As described in the introduction of this document, WP3.4 consists of two sub use cases with a different focus regarding their development subject leading to two different development V-models. In the Figure 7-1 these two V-model variants are sketched. The first variant (UC3.4a) has a strong focus about test case compilation and verification based on a set of requirements defined by a corresponding requirement model. The second variant (UC3.4b) deals especially with the development of the embedded verification platform to fulfil the requirement during the development. Both UC variants will be based on the use of simulation models for their corresponding purposes.
More than just being two separate instances of the common base UC, both variants should be combined as well in the following way: Since both use case variants are heavily based on simulation models, whereas the same, similar or related models are applied in both use cases (or are at least interrelated by certain simulation model parameters), the implementation of an appropriate Simulation Model Data Backbone (see brick No. B3.83, which is implemented in tight relationship with WP6.13) is essential to share certain models between the use cases. The simulation model data backbone should be attached to the appropriate tools in the several use cases as well under consideration of the principles of the interoperability standard (IOS).

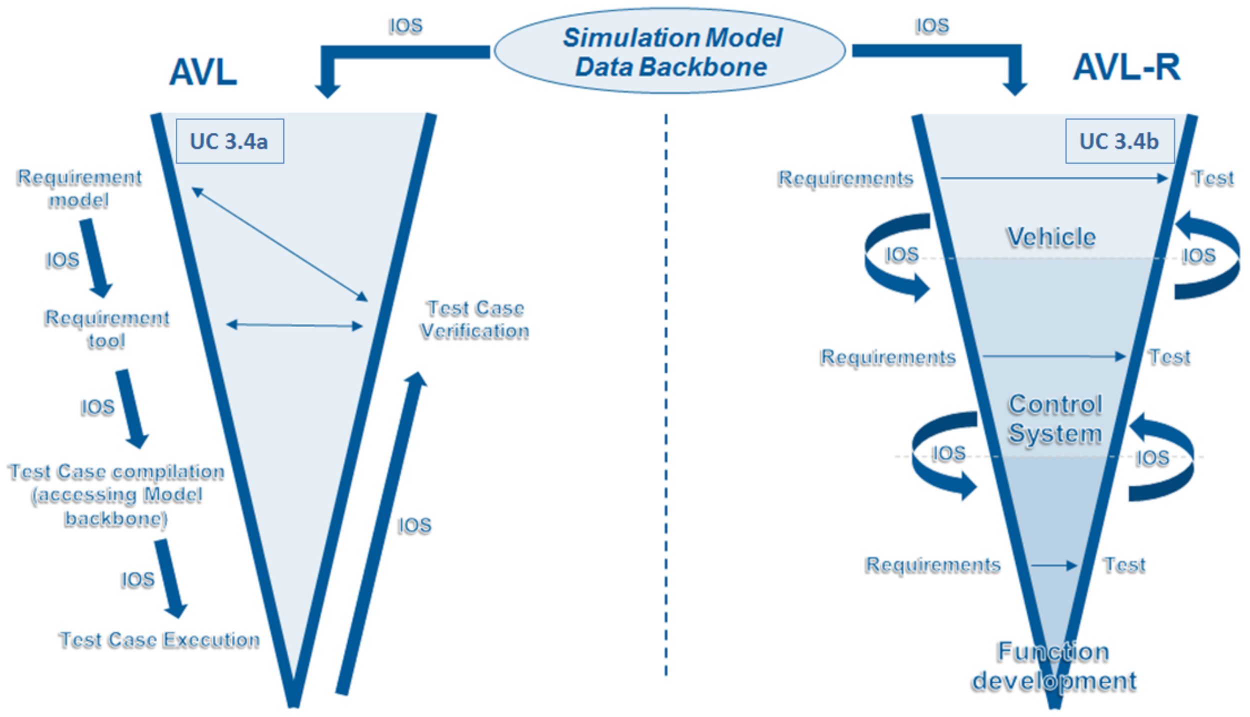


Figure 7-1: Model exchange between two different development process via IOS interfaces.

This chapter is structured as followed: In Section 7.1 further reflections regarding the motivations of combining different development V-models are described, whereas in Section 7.2 some concepts for feasible approaches under consideration of IOS are presented.

7.1 Motivation

The following scenario description details the major objectives described above and thus leads to the assignments that have to be accomplished by this work package:

1. Within a certain project and at a particular development stage, simulation models X are created for a specific purpose. These models describe several aspects of a certain object of interest (e.g. a mechanical part, the behaviour of a certain device, complex physical processes, etc.). The models are developed and embedded in certain development and simulation environments, whereas each environment may have its own execution semantics.

2. At different development stages of the same project or in an arbitrary development stage of another project, other simulation models Y are created with a similar or related purpose than simulation models X (see picture below). Due to the constraints, boundaries or requirements of the project or the development stage, the modelled aspects of Y may vary more or less significantly from the simulation model X, however, may also be comprised of overlapping or related aspects. Besides the deviation of the purpose of model X and Y, constraints such as simulation granularity and real-time requirements determine the model's content and simulation platform. Especially in this case, however, models X and Y may also be comprised of overlapping or related aspects.

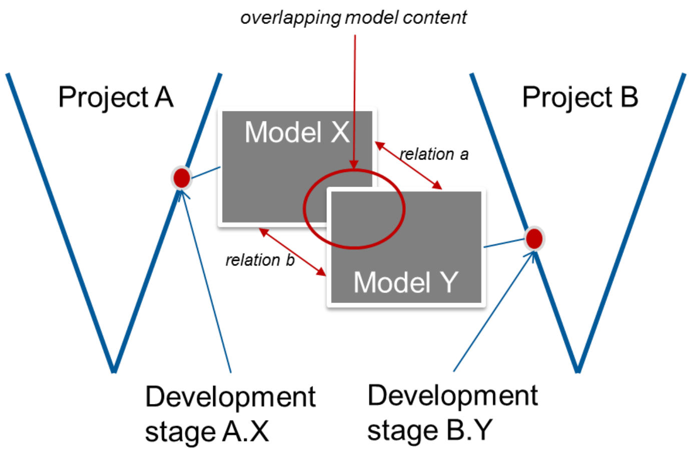


Figure 7-2 Interrelation between two models in two different development processes (may even developed and/or applied in different development phases)

3. As a consequence of 1) and 2) the following limitations would occur:
    a. Various model aspects (including the parameters and common variants of the modelled object) have to be re-modelled in every different project (or even development stage)
    b. Due to the lack of model sharing, the participants of the projects (or even the participants in the same project at different development stages) have limited or no access (and have thus as well often no awareness) of the model in other projects (or development stages).
    c. Even if the awareness is given, it is still a standard situation that interrelated data has to be exchanged manually with all the known limitations (e.g. lack of consistency, error-prone step, time efforts, etc.)
4. The CRYSTAL project will provide a significant contribution to the described limitation by providing an interoperability standard, which should provide a more straightforward access to the simulation platforms and its simulation models.
5. In addition to interoperability aspects, a centralized simulation model management environment (e.g. a simulation model data backbone) enables further perspectives to improve model development collaboration including the definition of relations between models.
6. Consequences of 4) and 5) are thus:
    a. Project collaboration becomes more straightforward (e.g. introducing a simulation model development categories and model development status across projects or development stages)
    b. An improved degree of automation (e.g. automatic synchronization of model data)
    c. A reduced set of development overhead (e.g. re-using of models)
    d. A standardized way how to access modelling data (e.g. introducing model categories, complying with interoperability standard)
    e. Enables the possibility of defining, exchanging and aligning of commonalities between models (e.g. model parameters)
    f. Should act as a door opener for further aspects such as versioning and variant management, applying requirement management, etc.

In the following section, a concept approach is presented with an IOS concept such as OSLC.

7.2 Towards an Interoperability Approach for Model Exchange between UC3.4a and UC3.4b via OSLC

In this section, an approach about simulation model exchange is presented. As sketched in the previous sections, different development processes are producer and consumer of models.

In case of UC3.4a for instance, simulation models for certain control systems (such as an engine control unit) are needed. These simulation models are provided by UC3.4b usually in form of appropriate Simulink models. As described as an engineering method e.g. in Section 8.6.3, these models may are used for co-simulation in an early testing phase in order to simulated the behaviour of an ECU in a particular vehicle, whose physical behaviour including its environment is (partly) simulated as well (e.g. for calibration iterations).

On the other hand, the control model needs to be tested as well, which is accordingly described by the V-model of UC3.4b. In order to test such functions without have the possibility to do that directly in a ready-to-use vehicle, vehicle simulations are needed. The AVL simulation tools Cruise and Boost are specialized tools for such kind of simulation models. These tools are applied at UC3.4a and are able to provide appropriate models for UC3.4b.

In Figure 7-3, a general interoperability pattern with the use of OSLC on top of various data backbones is presented. Let's assume that there is an in-house data backbone (called in AVL Data Backbone in case of WP3.4a), which stores various data categories assigned to the testing V-models of UC3.4a such as explained in Section 4.2.3. In addition, however, further data backbones have to be considered, e.g. provided by third party tools (such as PTC Integrity in case of UC3.4b). Depending on the data categories stored in these data backbones, a meta-model structure has to be defined on top of the data backbones that abstracts from the detailed content of each data backbone. This means in other words that these meta-model is defined independent of the location of storage on the one hand and is limited to only those data entities (across the data categories) that need to be linked to each other.

The OSLC concept perfectly matches to that demand. So-called OSLC domains capture a certain data category and OSLC resources reflect data entities that need to be linked against each other. It is important to mention that these OSLC resource models should be defined as minimalistic as possible and should thus be driven by the necessity of data relations. Every detail of a specific data entity should remain to the tool that creates and edits these details and/or the data backbone that is used to store these details.

It is the purpose of the OSLC adapters to map the concrete stored data entities to corresponding OSLC resources. As a consequence, the OSLC domains and resource models become independent of the belonging authoring tools and data backbones. Similarly, so-called uniform workbench is able to browse across the OSLC resource model, without the necessity of being aware of any data entity details: If such details are needed the workbench just delegates this to the corresponding authoring tool, e.g. by invoking the corresponding application via the OSLC adapter, which fetches the data from the data backbone and provide the data set to the invoked authoring tool.

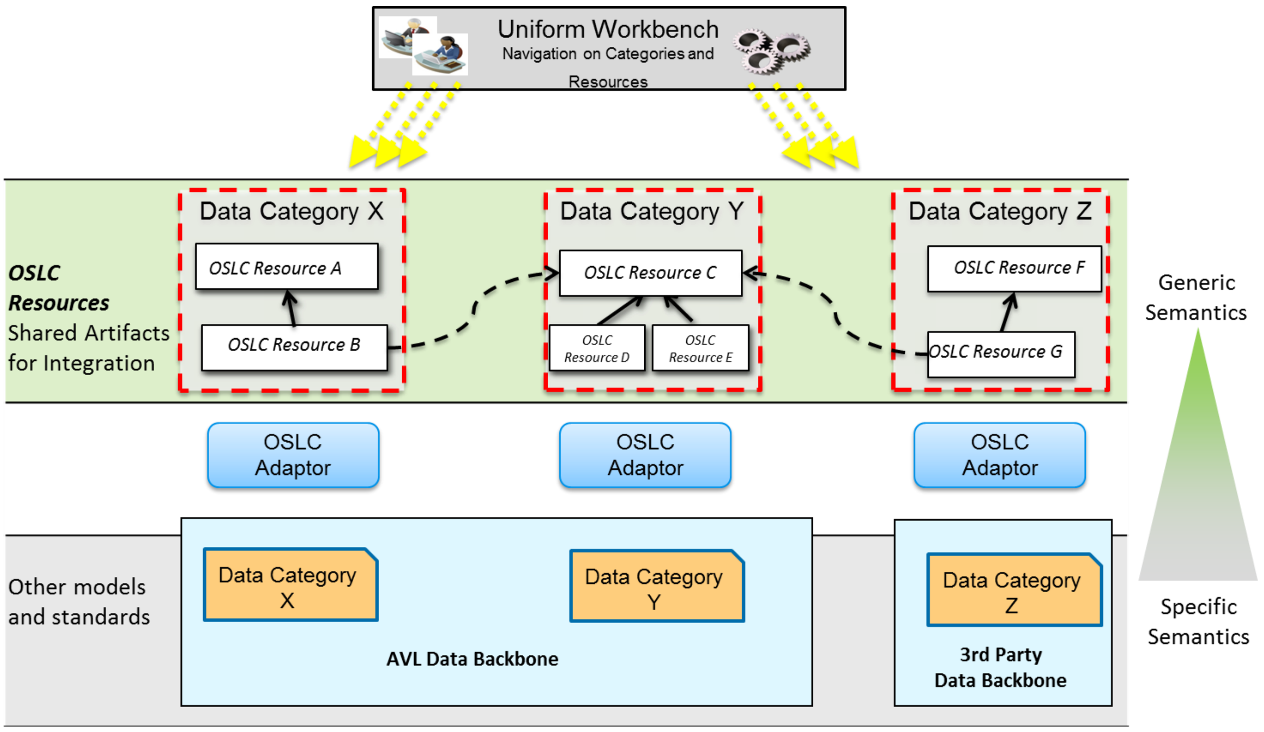


Figure 7-3: General OSLC pattern of the interconnecting data of different data backbones via OSLC

Figure 7-4 illustrates a possible OSLC architecture for exchanging simulation models between UC3.4a and UC3.4b. A possible implementation of the AVL Data Backbone is by using the AVL tool Santorin. Santorin is originally specialized on the management of measurement results of testing cycles and adheres to the measurement-related ASAM ODS standard. Since this tool is based on a database, it can be also used for the storage of other data entities (for the given example this would be for instance Boost simulation models needed by UC3.4b). In addition, the ASAM ODS provided possibilities of extending its data storage model if needed. In addition, Santorin comes up with a Navigator frontend the give support for browsing on the data entities stored in the database.

By introducing an OSLC architecture such as described before, the Navigator part of Santorin could now browse on a certain instance of the OSLC resource model. It could implement its own business model that reflects how to relate the different data set stored in the Santorin database. If details of the data are needed to be shown or edited, the Navigator just opens the adequate tool, which would be the built-in functionality in case measurement results and Boost in case of simulation models.

In addition, however, the Navigator would be also capable to browse on other data sources as Santorin. In UC3.4b PTC Integrity is used for data storage, which also holds the corresponding Simulink models for the ECU control software developed by this use case. The OSLC data structure thus enables establishing of relations across data backbones and development processes: For a specific project in UC3.4 a direct link to an ECU simulation model of UC3.4b could be established.

Correspondingly, from the UC3.4b point of view, the same mechanism can be applied: Needed vehicle simulation models (e.g. Boost models) developed in UC3.4a are referenced via corresponding OSLC links. The tool AVLab, used in UC3.4b, operates on the OSLC data structure (it has not to be the same instance as in UC3.4a) as well.

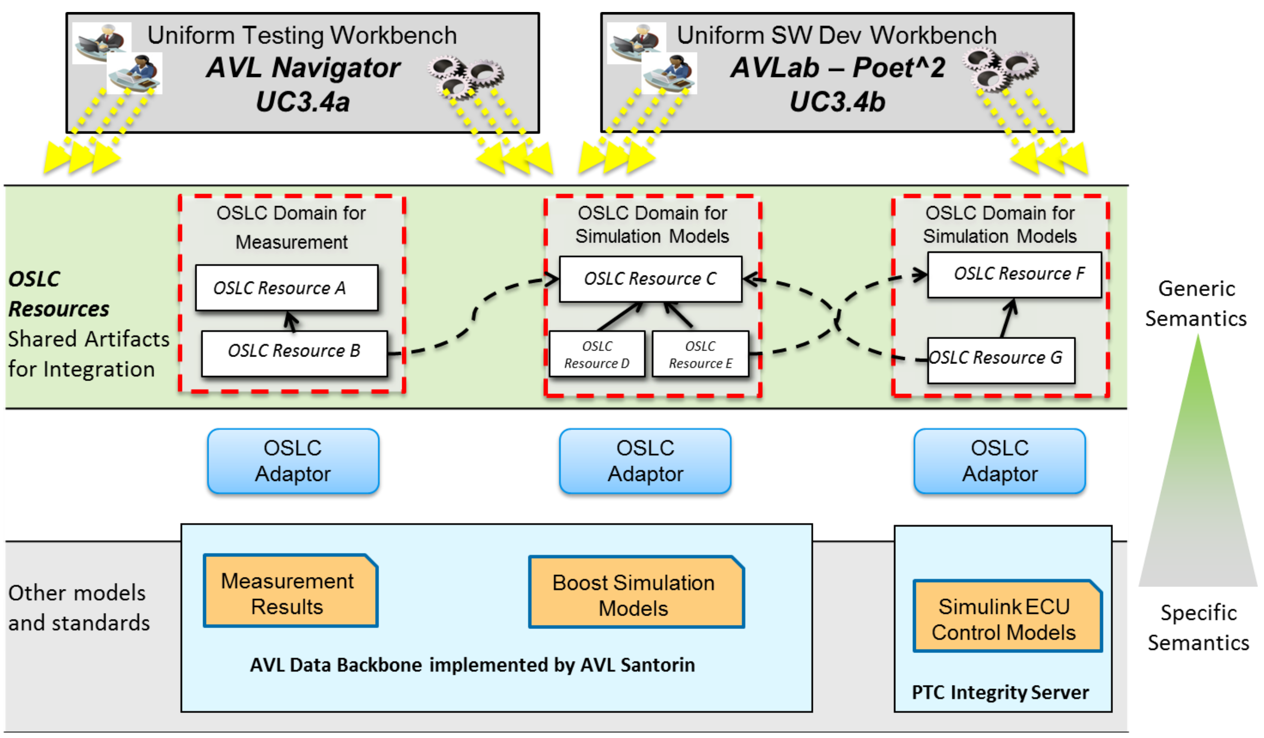


Figure 7-4 Possible OSLC architecture for exchanging simulation models

In Figure 7-5, a possible extension to the description above is illustrated: In UC3.4a requirements are stored in a corresponding requirement repository of the tool HP Quality Center. However, in UC3.4b this is done by the tool PTC Integrity. Based on the assumption that different OSLC data instances are used for UC3.4a and UC3.4b, requirements could be directly linked via the OSLC data structure to adequate models that fulfil these requirements. This would ease for instance the search of adequate models independent of the applied development process: E.g. by searching and browsing requirements of previous or parallel projects, possible simulation model candidates could be found more straightforward.

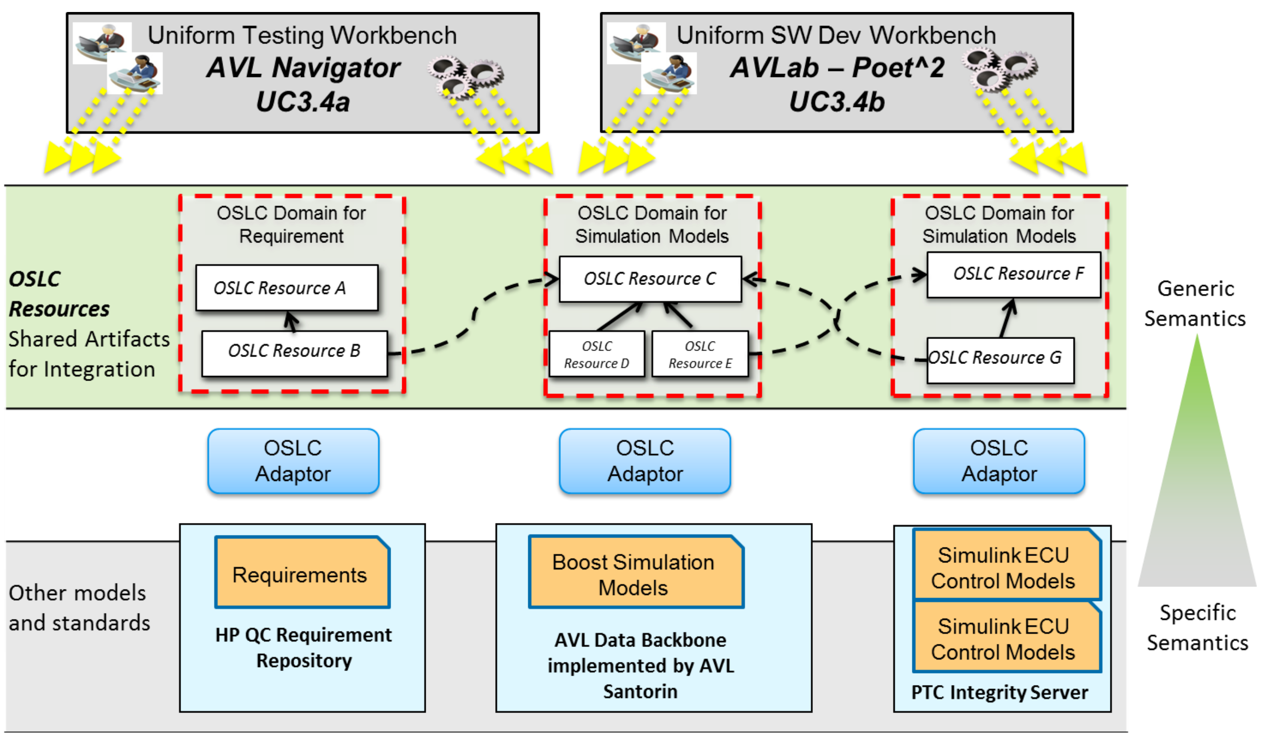


Figure 7-5 Possible OSLC architecture of exchanging simulation models based on a set of related requirements

8 Identification of Engineering Methods for UC3.4a

In this chapter, the derived engineering methods including their related activities, tools, data artefacts and their possible relation to the CRYSTAL IOS concepts are shown. Finally, a first draft version of an overall OSLC model is sketched, which needs to be refined during the projects by applying to corresponding tool chains. The content of this chapter is subject of change along the CRYSTAL project proceeds and thus represents the current snapshot of the use case specification.

8.1 Formalization of Requirements

The major purpose of this engineering method is deriving (semi-)formal, machine-processible requirements from natural language requirements. Currently at AVL, tools such as HP Quality Center or Excel are used to manage these natural language requirements. In UC3.4a we want to evaluate three methods to formalize them:

- Using SysML RM profiles to model requirements in SysML
- Using sequence chart based techniques to model some of the requirements (also in SysML/UML)
- Applying Boilerplate techniques

For these three approaches AVL works together with Fraunhofer IESE (SysML RM profiles), Chalmers (sequence chart based techniques) and ViF (Boilerpate). Purpose of applying three approaches is to learn about the benefits and drawbacks of each method in order to be applied in practise. For the boilerplate approach, ViF will develop a prototype tool, whereas for the requirement models Artisan studio should be used.

As illustrated in the Figure 8-1, the mentioned data artefacts need to be linked against each other. For that purpose, OSLC adapters should ensure traceability.

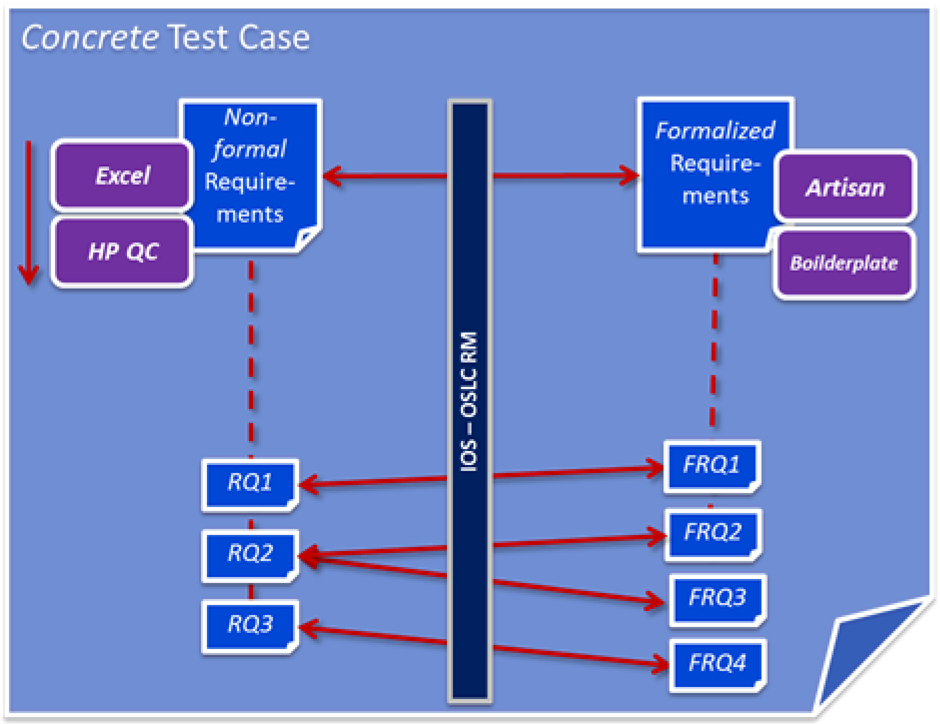


Figure 8-1: Links between textual requirements and (semi-)formalized requirements

8.1.1 Engineering Activities

Precondition for the engineering activities listed below is that natural language requirements are available and stored in a requirement management tool (e.g. HP QualityCenter). As a special activity in UC3.4a selected parts of the WLTP emission legislation define some of the requirements that should be formalized accordingly.

The *engineering activities* are then comprised of the following steps:

1. Reading the natural language requirements from natural language requirements tool into a requirements formalization tool (e.g. boilerplate).
2. The requirements engineer derives semi-formal textual requirements by means of the refinement tool. It can be the case that one natural language requirement is represented by n semi-formal textual requirements.
3. The requirements are validated against consistency, completeness, correctness, redundancy and unambiguousness.
4. The semi-formal textual requirements are stored in the requirements management tool.
5. Each semi-formal requirement gets linked to its corresponding natural language requirement.
6. The requirement engineer creates requirement models based on the semi-formal textual requirements (e.g. SysML-models in Artisan Studio).
7. The requirement models are stored in the modelling tool.
8. The requirement models get linked to their corresponding semi-formal requirements.

Consequently, the post-condition of these steps is that the semi-formal textual requirements and requirement models are stored properly and are interlinked to accordingly.

8.1.2 Data Artefacts

The following data artefacts are accessed, used or created by the engineering method:

- Original requirements in terms of natural language requirements (referred by a requirement ID)
- *Boilerplate templates* which are used during the formalization process
- *Ontology database* in order to use the right terms during the formalization process
- *Formal textual requirements* (boilerplates) with defined semantics for concrete *numerical values* as isolated numerical attributes as well as for *physical units*
- Requirement models in SysML/UML

8.1.3 Applying to the use case scenario

In the introducing Section 3.2.1.3, several testing requirements derived from the WLTP draft are defined. These requirements have to be formalized as described by Figure 8-2, Figure 8-3 and Figure 8-4 by using a so-called boilerplate approach. Boilerplates are requirement templates in order to standardize the expression of requirements to make them machine-processible.

Additionally, other formalization approaches such as the use of state diagrams to express these WLTP requirements enables even more: If a certain set of input vector of a particular test-run is known, these test-run can be tested against its compliance to the WLTP standard. Once this test has been positively evaluated and due to the fact that requirement traceability is ensured, the whole iteration within a testing V-model is evaluated to adhere to the WLTP specifications. More than that, if something changes within the WLTP specification itself, impact analysis on existing and formally used test-runs can be performed.

Natural language requirement:

RQ1. First gear shall be selected 1 second before beginning an acceleration phase from standstill with the clutch disengaged. Vehicle speeds below 1 km/h imply that the vehicle is standing still.

Boilerplate requirement:

Boilerplate:

If <condition> and <condition> and <condition>, the <system> shall <action> <parameter> within <time> <unit> before <state>.

RQ1. If <vehicle speed less than 1> and <clutch disengaged> and <beginning an acceleration phase>, the <test equipment> shall <select gear> <gear 1> within <1> <sec> before <acceleration phase>.

Figure 8-2: Formalized WLTP requirement RQ1

Natural language requirement:

RQ2. Gears shall not be skipped during acceleration phases. Gears used during accelerations and decelerations must be used for a period of at least 3 seconds.

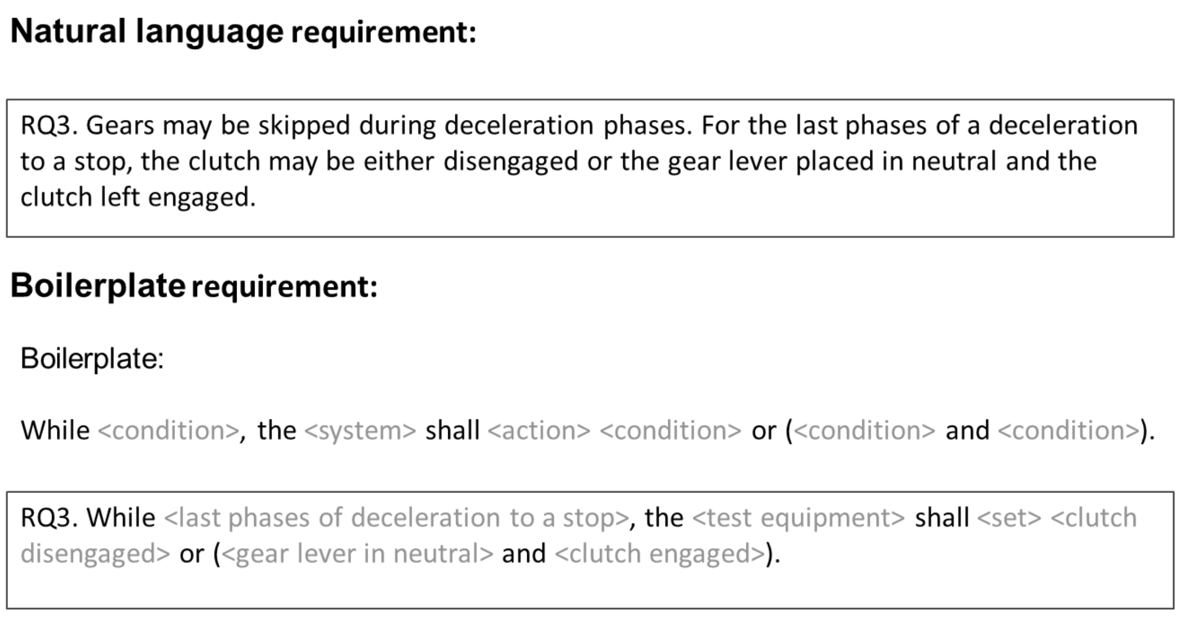
Boilerplate requirement:

Boilerplates:

While <condition>, the <system> shall <action> with <entity> less than <quantity>.
While <condition> or <condition>, the <system> shall <action> for at least <time> <unit>.

RQ2.
While <acceleration phase>, the <test equipment> shall <shift gear> with <gear difference> less than <1>.
While <acceleration> or <deceleration>, the <test equipment> shall not <change gears> for at least <3> <sec>.

Figure 8-3: Formalized WLTP requirement RQ2


Natural language requirement:

RQ3. Gears may be skipped during deceleration phases. For the last phases of a deceleration to a stop, the clutch may be either disengaged or the gear lever placed in neutral and the clutch left engaged.

Boilerplate requirement:

Boilerplate:

While <condition>, the <system> shall <action> <condition> or (<condition> and <condition>).

RQ3. While <last phases of deceleration to a stop>, the <test equipment> shall <set> <clutch disengaged> or (<gear lever in neutral> and <clutch engaged>).

Figure 8-4: Formalized WLTP requirement RQ3

8.2 Verification of Requirements

Purpose of this engineering method is the verification of requirements against a specific *test results* (e.g. *measurement results*) of a specific *test case* as illustrated in Figure 8-5. At this point of the engineering method, it is not distinguished between natural textual requirements and (semi-)formalized requirements. Thus the tools on the requirement management side are *HP QualityCenter*, *Artisan Studio* and the *boilerplate prototype application*. The requirements defined by these tools are verified by concrete test cases that lead to specific measurement results. *AVL Santorin* is the tool of choice in this use case to manage and access these measurement results. The actual process of requirement verification is captured by the AVL VeVaT tool. Due to the availability of proper data artefact links, AVL VeVaT is able to compare the measurement results against the formalized requirements and finally verifies or falsifies the associated natural language requirement. If some measurement value post-processing is needed for that step, AVL VeVaT instruments AVL Magic, which is designed for such tasks.

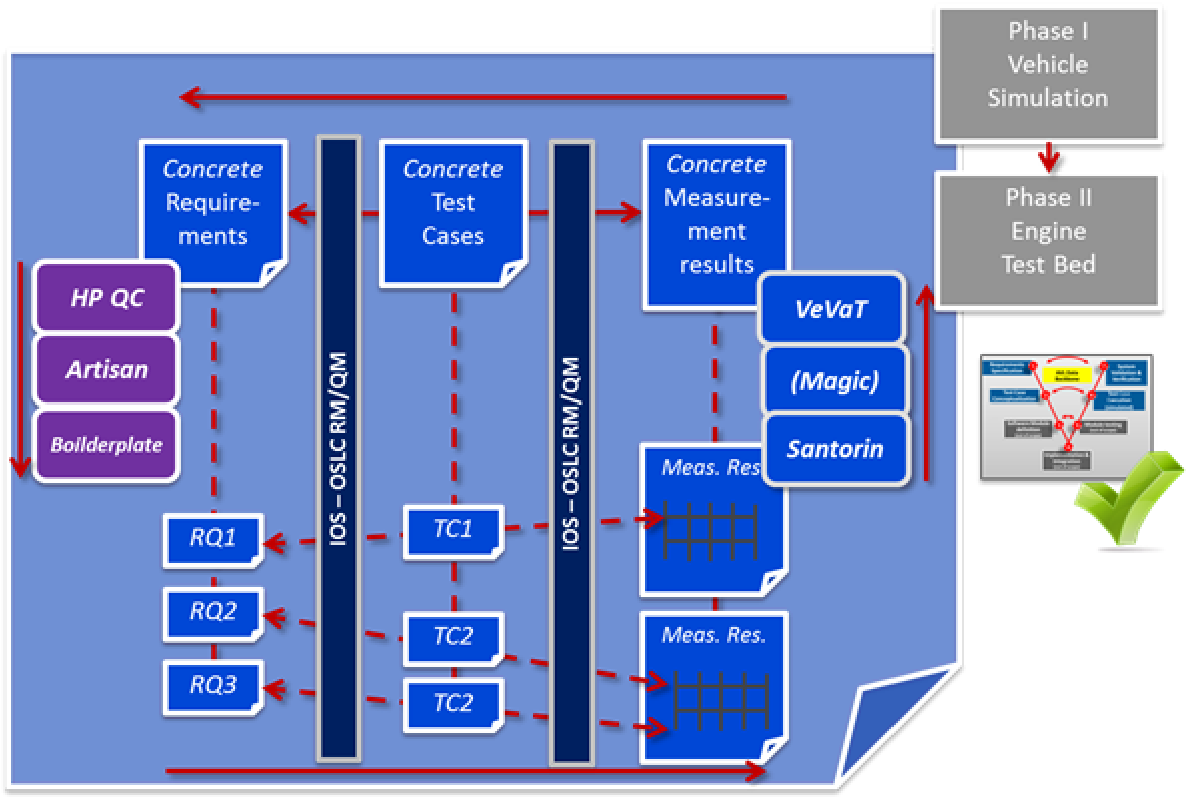


Figure 8-5: Establishing links between requirements, test cases and measurement results

8.2.1 Engineering Activities

The *pre-condition* of for this activity is that the requirement formalization activity has been finished and the (formalized) requirements are stored and accessible. Furthermore the associated test cases have been executed and the test results (e.g. measurement values) are stored and accessible as well.

The *engineering method* is now comprised in more detail of the following activities:

1. Requirements (formal and natural language) are read from the requirements store or tools into the AVL VeVaT requirement verification tool
2. The linked test cases and the UUT-related measurement data are read
3. AVL VeVaT analyses the measurement data (e.g. via AVL Magic) and compares them to the formalizes requirement limits
4. The verification results (i.e. passed or failed for all checked requirements) are generated – optionally with some addition information (such as failure reason, detected deviation from limits. etc.)
5. The verification results are stored as annotations in the requirement management tools

8.2.2 Data Artefacts

The following list summarizes the occurring data artefacts:

- *(Verified) semi-formal requirements* with properties such as requirement IDs, numerial limits, physical units and additional pass/failed flag with an optional description
- *Validation rules* comprised of formulas, scripts, algorithms which are called for analysing the test results (measurement data) and compare them to the formalized requirements
- *Test cases* are containers with references to further data artefacts that are needed to execute concrete test runs, which produce then concrete test results.

- *Test results (measurement data)* are usually time-based numerical data channels with channel names, units, sequence of numeric values (standardized by ASAM ODS).

8.2.3 Applying on the use case scenario

In Figure 8-6, a possible data structure of the relations between requirements, test cases and measurement results is sketched, which adhere to the idea of our use case scenario: The requirement about emission limitation has some relations to a concrete test case that should verify this requirement. The test case itself consists of further data artefacts related to the use case scenario: Test runs defined by the WLTP, the injection time parameter, which is used here for calibration and finally the configuration for the sensor the measures the emission value, which is then verified against the requirement. This data structure has to be further evolved during the project duration.

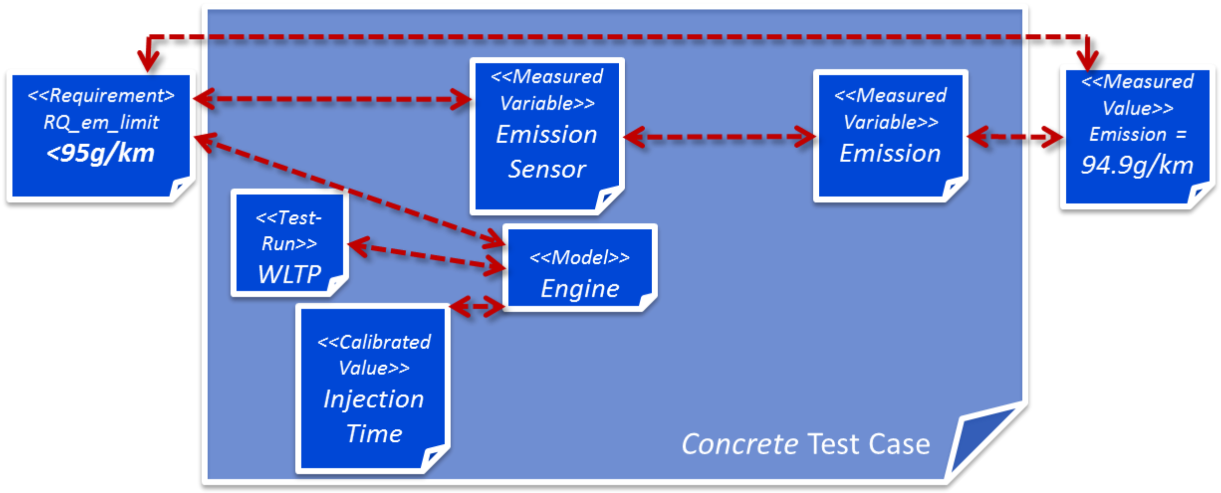

<<Requirement>> RQ_em_limit <95g/km
<<Measured Variable>> Emission Sensor
<<Measured Variable>> Emission
<<Measured Value>> Emission = 94.9g/km
<<Test-Run>> WLTP
<<Model>> Engine
<<Calibrated Value>> Injection Time
Concrete Test Case

Figure 8-6: Traceability across a requirement, a test case and a measurement result

8.3 Requirement Traceability to Authoring Tools

As already indicated by the previous engineering method in Section 8.2, requirements and test cases have to be linked against further data artefacts that are verifying the requirements and are executing the appropriate test cases. In our use case, for instance, this aspect is especially needed for simulation models. Simulation models play an important role in most of the testing activities and phases and exchanging models across development phases is a crucial factor. It would be a significant shift for supporting model re-use if traceability between requirements and models elements (that are about to verify these requirements) is established consistently.

In Figure 8-7, a proper scheme for related interoperability challenges is sketched: An authoring tool (e.g. Simulink, AVL Cruise, etc.) is connected to an ALM Environment (e.g. HP Quality Center, PTC Integrity) via OSLC adapters. On top of these adapters an OSLC resource model addresses exclusively the top level elements such as requirements, test cases as well as models including their basic elements. Everything else remains to be interpreted by the authoring/ALM tool itself.

This engineering method is a relevant foundation for the engineering method about verification of requirements as described in Section 8.2.

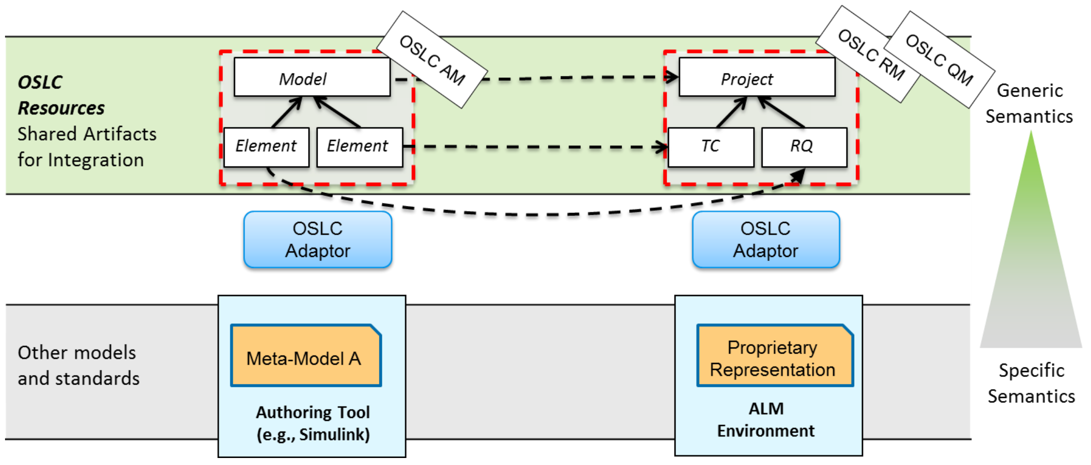


Figure 8-7: Towards an IOS OSLC scheme for interlinking authoring tools and requirements

8.3.1 Engineering activities

Common engineering activities for establishing appropriate traceability would be:

1. Creating a new project and setting requirements and test cases
2. Analysing a relevant set of requirement and test cases against their coverage rate by checking if every selected requirement and test case is covered by a corresponding simulation model or simulation model element.
3. If requirement coverage is not sufficient then assign a simulation model and/or test case to a affected requirement
    a. Searching for simulation models and test cases that are already assigned to similar requirements (e.g. from previous projects)
    b. If no simulation models are found define new ones new ones.

8.3.2 Data artefacts

Due to its relation to engineering method about verification of requirements, see Section 8.2.2 for involved data artefacts. Test cases usually act as a kind of container for further data artefacts. Note that in Figure 8-8 simulation models are also among those artefacts. In addition, however, they should also have a direct link to the associated requirement. Even more, it should be also possible to link requirements against sub-elements of such simulation models as illustrated in Figure 8-8.

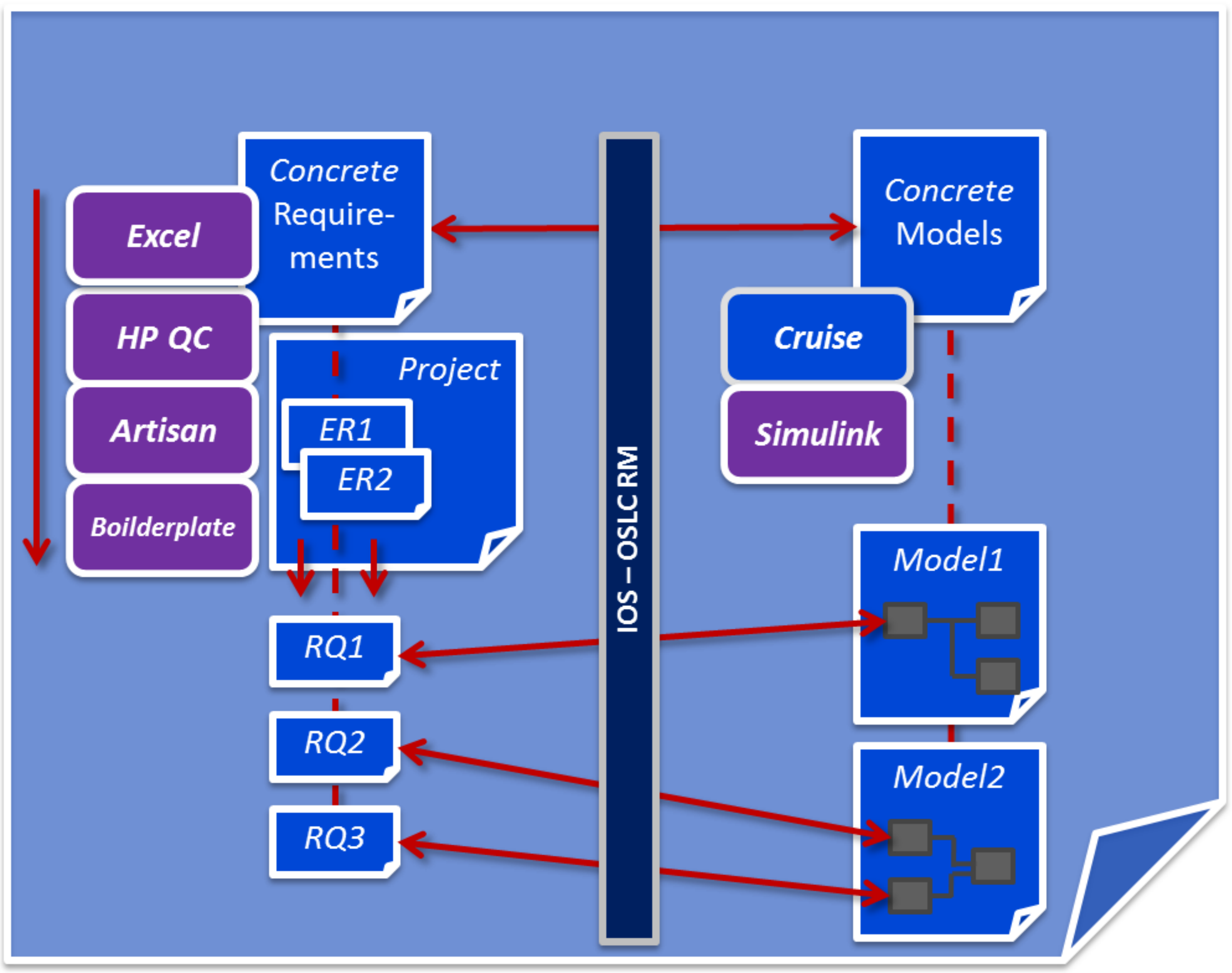


Figure 8-8 Linking requirements against simulation models and sub-elements of these models

8.3.3 Applying on the use case scenario

Due to its relation to engineering method about verification of requirements, see Section 8.2.3 for more information and note especially Figure 8-5, whereas a link from the emission limitation requirement to a simulation model is set.

8.4 Test and Calibration Iterations

In Section 4.1.2, the test and calibration pattern was initially described: Based on the proper definition of requirements test and calibration iterations are performed as illustrated once more in Figure 8-9. The pattern consists of five parts called requirement definition, calibration set-up, testbed and UUT set-up test-run simulation/execution and iteration results.

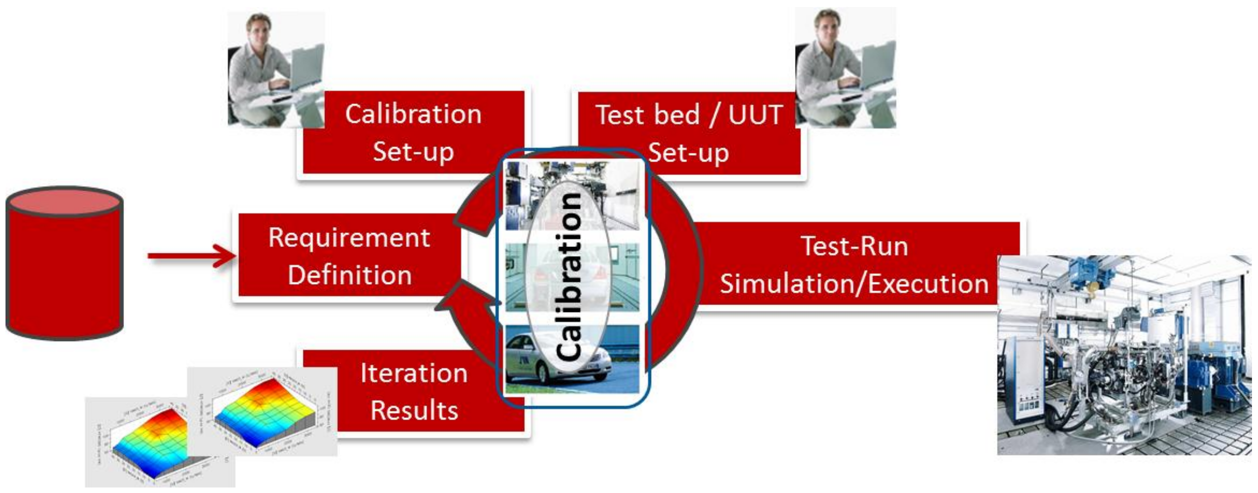


Figure 8-9: The test and calibration pattern

As illustrated in Figure 8-10, four of these five parts (except the test-run simulation/execution; see middle part of the figure) can be naturally associated to our five data categories (upper part of the figure) as defined in Section 4.2.2 as well to concrete data artefacts as shown in Figure 8-10.(lower part of the figure).

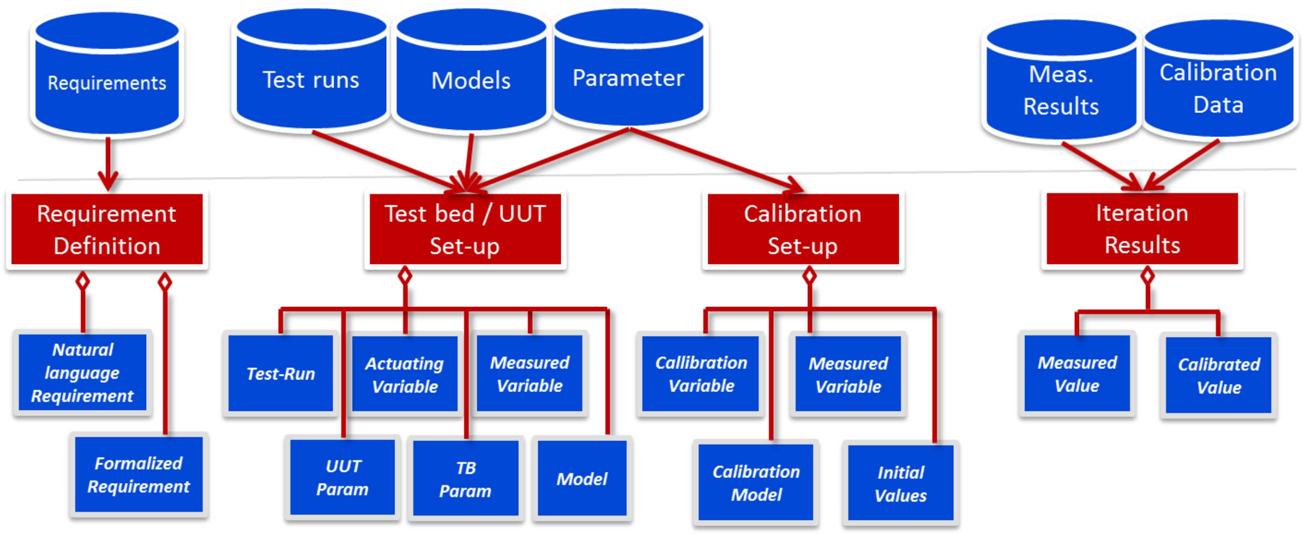


Figure 8-10: Associate Test and Calibration pattern parts to data categories and data artefacts

The requirement definition and verification activities were already described in detail by the corresponding engineering methods in the Sections 8.1 and 8.2 and thus capture the questions how the requirements are defined and how the iteration results are applied on the requirement verification process.

This engineering method in this section explains the activities between requirement definition and verification. In Figure 8-11, these activities are sketched for a test and calibration iteration during testing phase I (vehicle simulation), which is entirely based on simulation. The next engineering method in Section 0 explains also the test and calibration pattern for testing phase II (engine test bed) in order to demonstrate data migration from testing phase I to testing phase II.

8.4.1 Engineering activities

Pre-condition for this engineering method is that requirements are well defined and formalized as well as linking between requirements and models has been established (as described by the engineering method is Section 8.1 and 8.3). In addition, access to data (e.g. test cases, requirements) of previous projects should be established in order to re-use initial data sets.

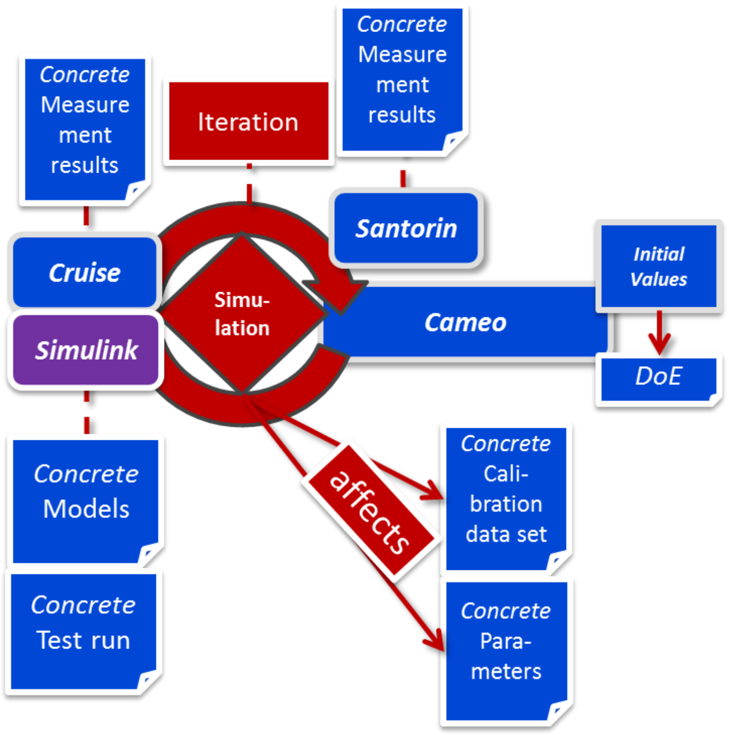


Figure 8-11: Test and calibration iteration in testing phase I (simulation)

The engineering activities inspired by the test and calibration iteration (including their artefacts) illustrated in Figure 8-9, Figure 8-10 and Figure 8-11 are the following.

1. Selection of the appropriate requirements that need to be fulfilled (see Section 8.1)
2. Selection of the appropriate simulation models that are linked to that requirements (see Section 8.3)
3. Configuration of the corresponding Calibration Set-up:
   a. Selection of calibration variables
   b. Selection of initial calibration values (e.g. from calibration models linked to similar requirements of previous projects)
4. Configuration of the corresponding test bed set-up (this task activity can be skipped in the testing phase I, since the whole test-run is based on simulation models):
   a. Selecting of existing test bed set-ups based on similar projects and requirements
   b. Creation of a new variant for this test bed setup
   c. Adaptation of the new variant according to the given requirements
5. Configuration of the corresponding UUT Set-up
   a. Configuration of the physical part of the UUT Set up such as UUT parameters (this task activity can be skipped in testing phase I)
   b. Configuration of the simulated part of the UUT such as simulation model parameters (rest-vehicle simulation)
      i. Creation of a new variant of existing simulation models of previous projects with similar requirements or simulation models used in previous testing phases.

   ii. Adaptation of the new created model variant according to the requirements and the remaining simulation parts of the rest-vehicle simulation

6. Performing the ((partially) simulated) test run
   a. Selection of the test run input values for the given test case (e.g. WLTP-based)
   b. Execute the corresponding test case
7. Collecting and analysing the test run results against requirements as described in more detail by the engineering method in Section 8.2
8. Applying calibration methods against requirements
   a. Creating or adapting a calibration model-based on the test run results
   b. Adjusting calibration variables against requirements
   c. Roll-out calibration variables to associated parameters
9. Start next test and calibration iteration as described in paragraph 4.

Post-condition is then a fully calibrated and parameterized UUT that should fulfil the given requirements. The whole set of data artefacts should act as the basis for a test and calibration iteration in the next testing phase, whereas some parts of the (rest) vehicle simulation is replaces by a physical UUT.

8.4.2 Data Artefacts and Data Model

The most important data artefacts that are consumed and produced during the current engineering method are already described in the previous sections especially by Figure 8-10. Therefore they will be not described once more in this section. However, in this engineering method the necessity of data relations that adhere to a data model becomes mandatory: without such data relations, the activities of this method will not be successful. In Section 3.1.3 it is suggested that these interoperability challenges are mastered by introducing a high-level data model based on the OSLC linked-data approach.

In Figure 8-12, such an approach is sketched for calibration data management: A calibration tool (e.g. AVL Cameo) is storing its data in a proprietary format, but is also capable of exporting some aspects of this data to different standardized formats (such as ASAM MCD-2 MC for calibration data and ASAM ODS for measurement results). The exported data may be stored in a central data base such as the AVL data backbone concept (e.g. using AVL Santorin). OSLC adaptors on top of all involved tools and data providers, however, allow direct access to top level elements of the data artefacts applied in the engineering method about test and calibration iteration. These top level elements are elements of a high level OSLC resource model that complies with artefacts presented by this engineering method. A uniform workbench (such as the AVL Navigator tool) can then be used to navigate on a concrete instance of this OSLC model. Consequently, if the activities (of which the current engineering method is comprised) are embedded in this workbench properly, the usage of this tool would ensure that links are set properly and enables corresponding data navigation and reuse in related projects or later testing phases.

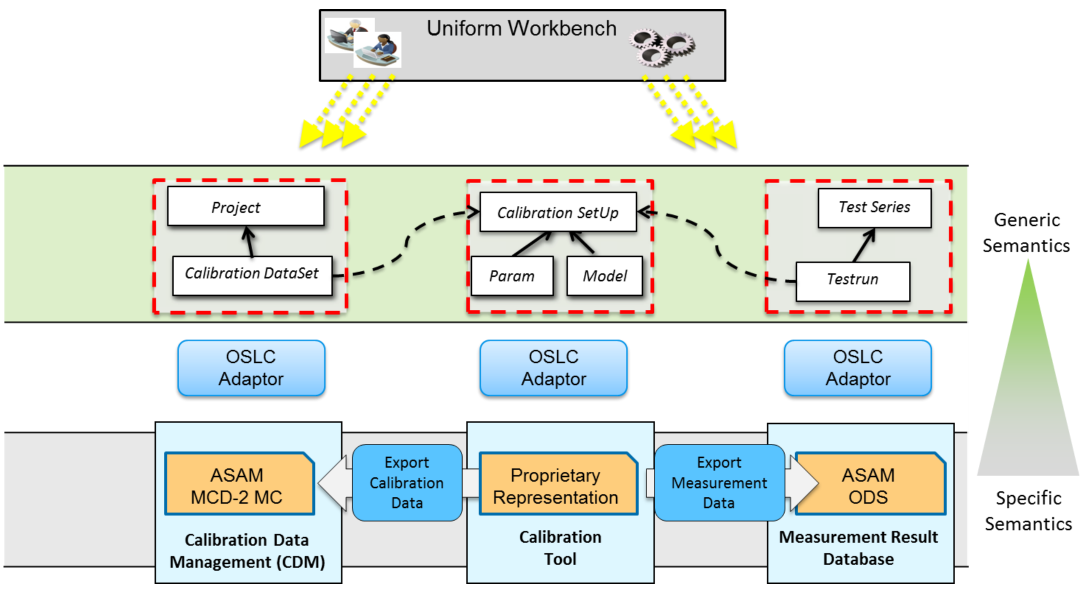


Figure 8-12: OSLC linked-data approach for calibration data management

8.4.3 Applying on the use case scenario

An example for interlinked data artefacts (based on the requirement of our use case scenario) based on an OSLC model is sketched in Figure 8-13. It is part of the project to find and define the best suited data model.

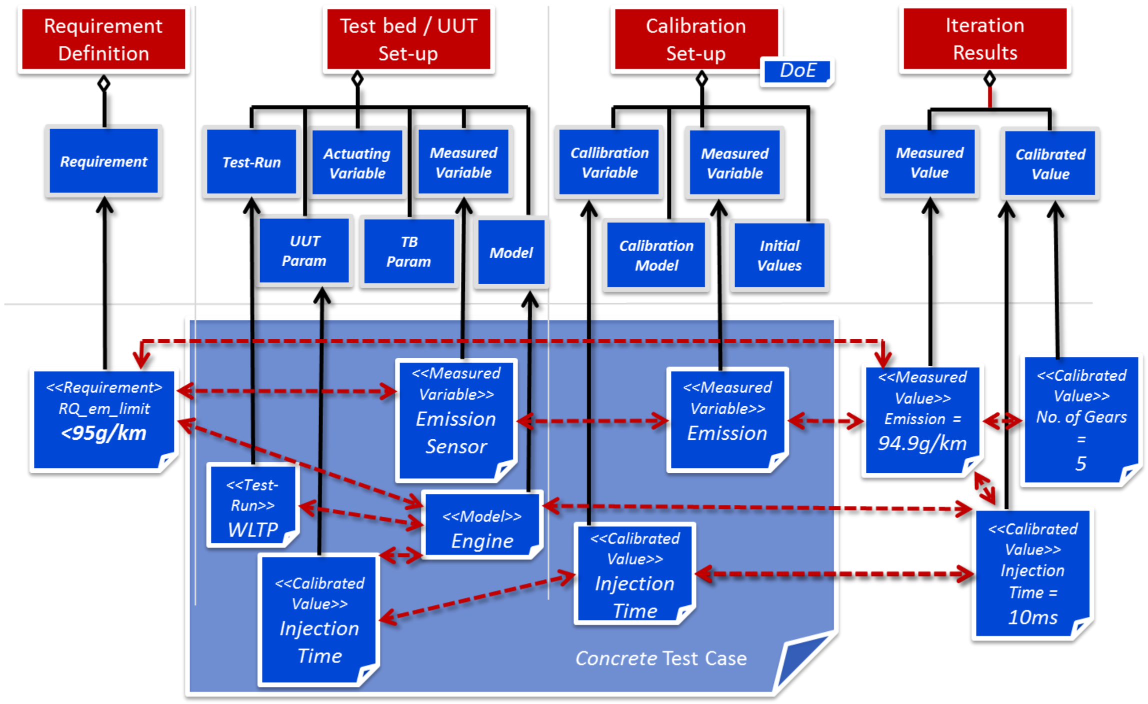


Figure 8-13 Example of interlinked data artefacts

8.5 Data Management across Development Phases

The previous section describes the engineering method about the test and calibration pattern with examples from the testing phase I (vehicle simulation). In this section, a closer look to data migration from one testing phase to another – one of the key issues of UC3.4a – is presented especially for this application pattern. Figure 8-14 illustrates an example for such a scenario for the test and calibration patter especially regarding data migration from testing phase I (vehicle simulation) to testing phase II (engine testbed).

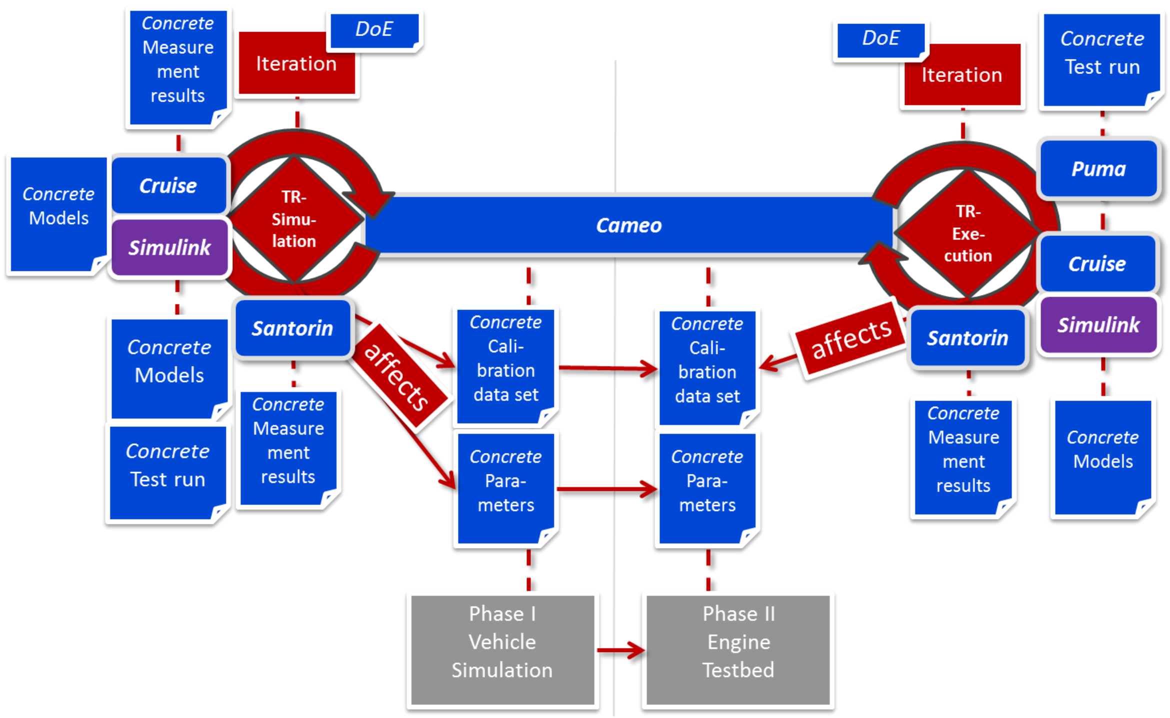


Figure 8-14 Data migration from testing phase I to testing phase II

The testing and calibration iteration is complemented by another tool called AVL Puma, which is the central control software for AVL test beds. On a first glance, the major difference between the phases belongs just to the essential fact that a former virtual component (i.e. the vehicle engine) is replaced by a physical component (embedded in a test bed set-up). A consequence of that is, that physical parameters (such as the cylinder capacity) are not part of the calibration iteration any more. In other words, the number of calibration variables is lower in phase II than in phase I.

The remaining data artefacts seem to be the same in both phases. However, there are number of serious constraints that make things not that straightforward as they might look like. The two most prominent constraints are:

- In many cases, projects that belong to testing phase I and testing phase II are de-coupled and performed by different project teams with separated data repositories. In addition, if one project in testing phase I has been finished, the continuing project for testing phase II is not started immediately. Due to the de-coupled projects parameters, calibration data and measurement values are often not consistent in their semantics.
- In the different development phases, different simulation models have to deal with different demands. In some cases, accuracy is of highest important, which leads to long simulation durations. Especially if physical components are part of the testing procedure (such as in testing phase II), real-time constraints have to be considered. This leads to different representations of models and hinder a 1-by-1 re-use. However, this does not mean that there is not a great potential in ensuring semantic consistency between these models.

8.5.1 Engineering Activities

In Figure 8-15, some typical project-related activities are illustrated as the usually happen in testing phase I (vehicle simulation).

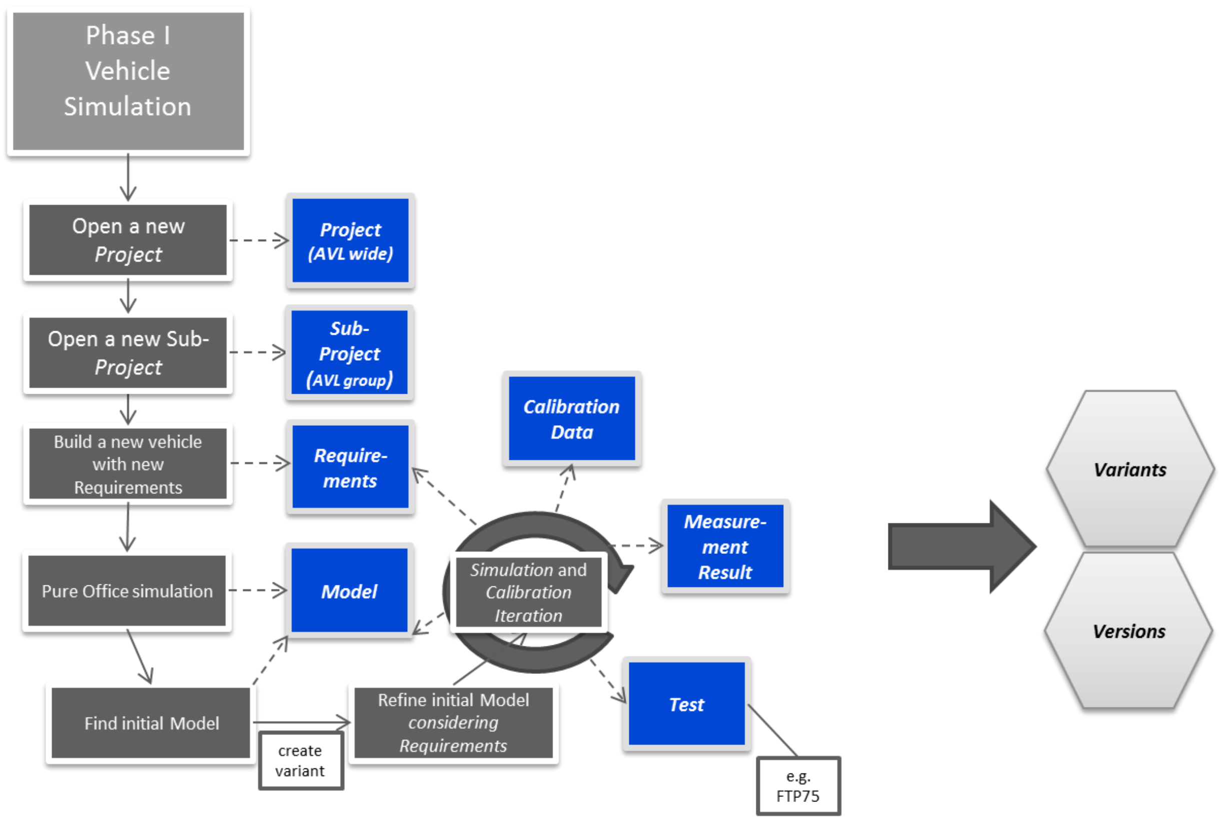


Figure 8-15: Typical project activities and data artefacts in testing phase I

Let's assume that a customer wants to build a new vehicle. Then the engineering methods are usually comprised of the following activities:

1. Open/Create a new project for the new vehicle. This project should be defined globally and independent of the development/testing phase and the project team
2. Open a new Sub-Project for the specific development/testing phase usually assigned to a specific project team
3. Collect and define requirements for the new vehicle (eventually based on former requirements of similar projects)
4. Based in these requirements perform a pure office simulation
    a. For that find an initial simulation model from a similar project associated with similar requirements
    b. Create a new variant of this simulation model
    c. Refine the initial model considering overall project requirements.
5. Perform the simulation and calibration iteration as described in Section 8.4.

In Figure 8-16, the data re-use from testing phase I (vehicle simulation) in testing phase II (engine test bed) is illustrated with the following activities:

1. Based on the assumption that the development of the same vehicle started in the previous testing phase I should be continued, the associated overall vehicle project is reactivated.
2. A new sub-project associated with the actual testing phase is added.
3. The simulation models from the previous phase are re-used as initial vehicle models for the actual testing phase.
4. Where necessary, new variants of the simulation model are created. A typical variant for testing phase II (engine testbed) is to replace the engine model of the overall vehicle simulation model by corresponding interfaces to the testbed control system. This testbed control systems consists of a physical engine.
5. Associated test case data artefacts (calibration data, measurement results, test-runs) are allocated form the previous testing phase and are re-used or mapped to the actual testing environment.
6. For the test bed itself, an initial test bed configuration is selected from previous and similar projects.
7. A new variant of this test bed configuration is created.
8. This new variant needs to be adapted according to the actual requirements and UUT parameters (i.e. engine parameters).
9. Testing and calibration iterations are performed as described in Section 8.4.

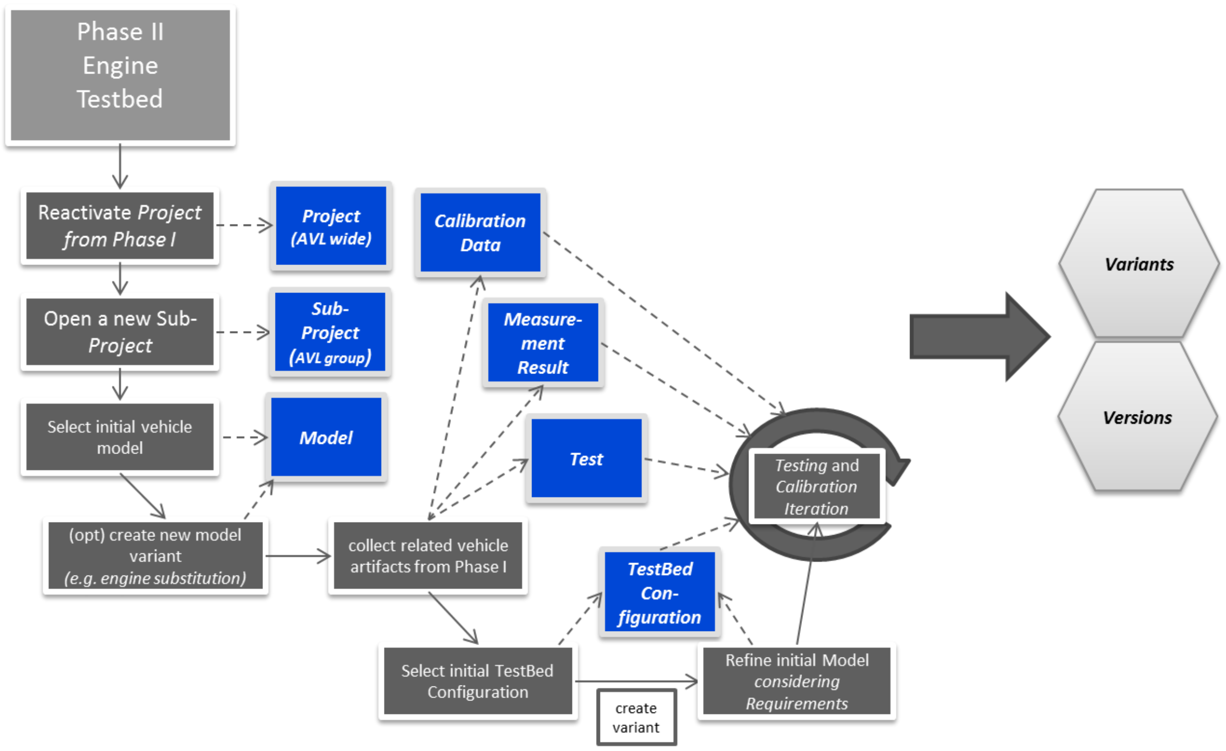

Phase II Engine Testbed
Reactivate Project from Phase I
Project (AVL wide)
Calibration Data
Open a new Sub-Project
Sub-Project (AVL group)
Measure-ment Result
Select initial vehicle model
Model
Test
(opt) create new model variant (e.g. engine substitution)
collect related vehicle artifacts from Phase I
Testing and Calibration Iteration
TestBed Con-figuration
Select initial TestBed Configuration
create variant
Refine initial Model considering Requirements
Variants
Versions

Figure 8-16: Typical project activities and data artefacts (including data re-use) in testing phase I

8.5.2 Data Artefacts

Many of the needed data artefacts categories for data management across development phases are already illustrated in the activity diagrams above. Besides re-using these data artefacts in different testing phases, their interrelations need to be standardized across these testing phases. This could be done by a corresponding meta-model, e.g. by a corresponding OSLC resource model. In Figure 8-17, a first reasonable draft is sketched. This draft needs to be evaluated and adapted according the research insights gained during the CRYSTAL project.

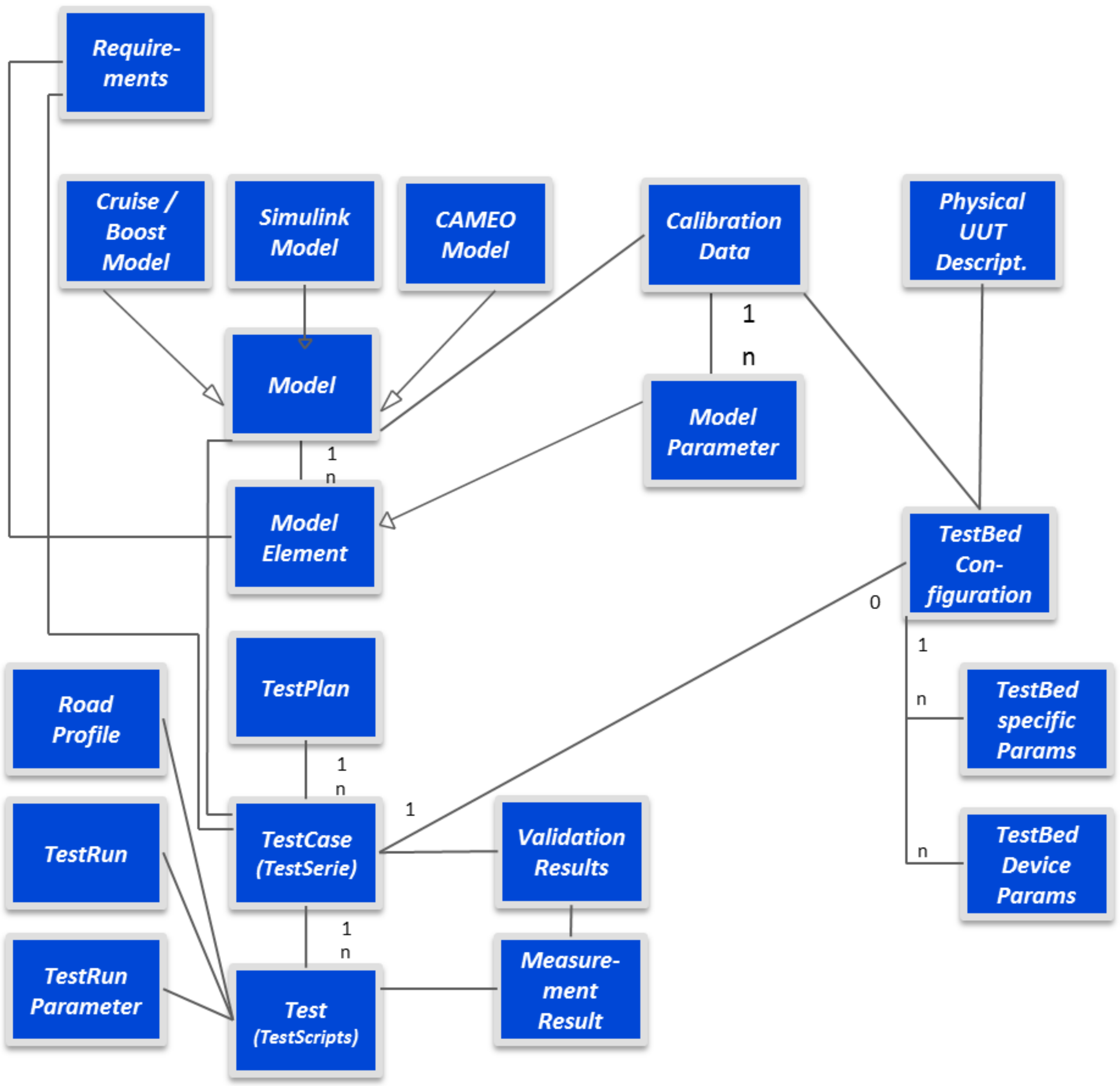


Figure 8-17: Towards a project-independent data model (common data artefacts)

8.6 Heterogeneous Simulation

Another example of combining OSLC with other standards is illustrated in Figure 8-18. Heterogeneous simulation (also called co-simulation) is the ability to couple two or more simulation models executed in different tools at run-time. The FMI standard currently evolves itself to be *the* standard for co-simulation (so far there has been none). This kind of interoperability has by nature nothing to do with the linked-data concept of OSLC. Nevertheless, a useful combination of these interoperability types is possible: Independent of the run-time aspects and details such as which data ports of the models have to exchange data, the given fact that two models are related to each can be defined by OSLC links as well. Corresponding OSLC adapters abstract from the particular modelling tools and map the corresponding model to a OSLC resource model, which consists just of basic standard modelling elements representing a hierarchically structure and which model elements of a model A are related to what model elements of model B. Details of these model-interrelations such as the applied data types of concrete connected ports may remain on the responsibility of the FMI co-simulation standard.

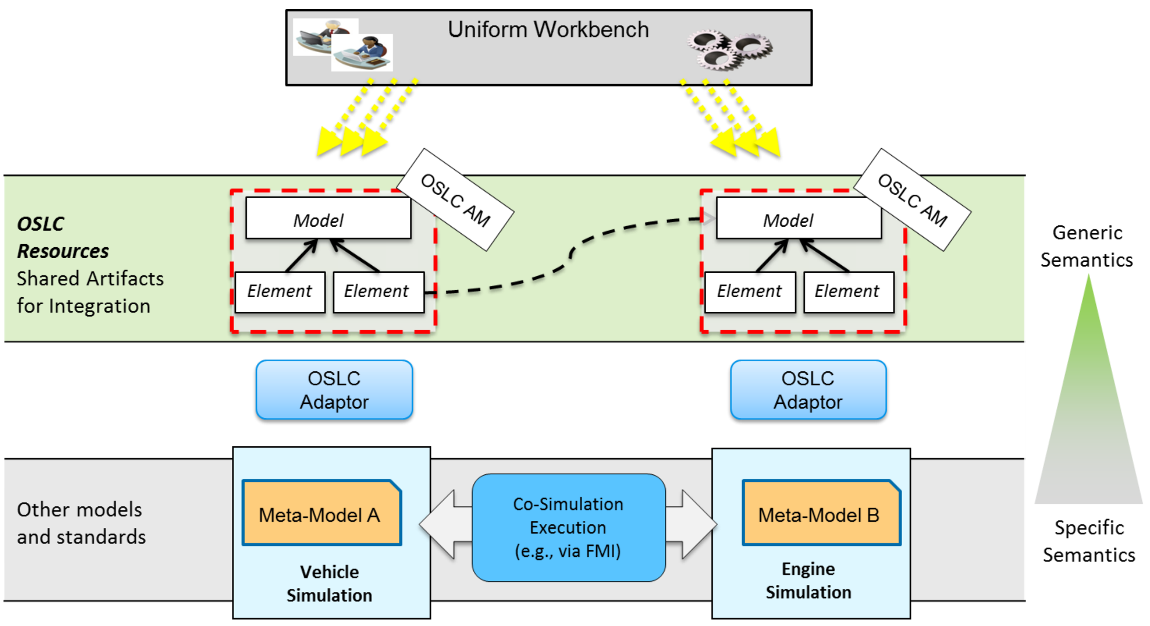


Figure 8-18: Combining the OSLC linked-data approach with co-simulation aspects

Figure 8-19 illustrates an example from the automotive domain: A rather overall vehicle model, which is embedded in an adequate vehicle environment model, and a more detailed engine model should be co-simulated. Two different simulation tools may be involved here, whereas each is specialized for its domain. For instance, the motor model is about a software model that consists about the control algorithms of the engine's ECU.

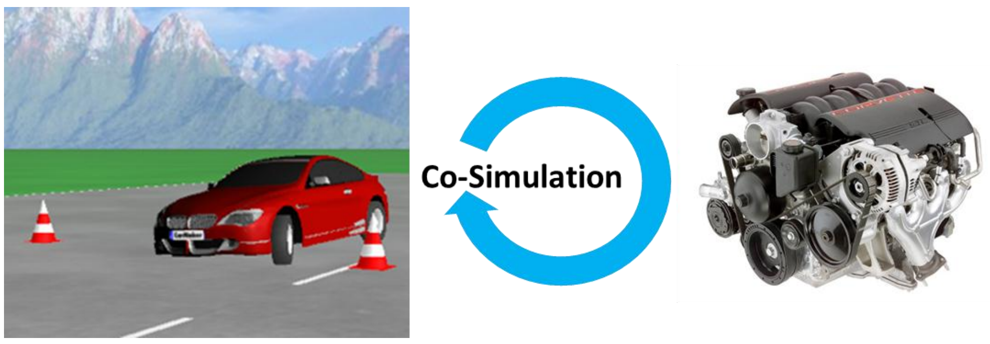


Figure 8-19: An automotive example about the co-simulation of an overall vehicle and vehicle environment model and a more detailed engine model

8.6.1 Engineering activities

Engineering method in case of co-simulation is comprised of the following activities:

1. Based on the given requirements, the most adequate simulation models are selected from a model library as described by the engineering method in Section 8.3 (e.g. data backbone, models that are linked to similar requirements, etc.)
    a. If available, the models need to be modified or configured according the given requirements
    b. If not available, the appropriate simulation tool has to be selected
    c. If the selected models are based on the same simulation tool, no co-simulation is needed and thus the next steps need not to be considered

2. The selected simulation models have to be analysed regarding their type and relation to each other
3. The model relations need to be documented, e.g. by high level OSLC link relations.
4. Communication interfaces (data ports) according to the principles of FMI have to be defined.
5. Data ports have to be analysed regarding their properties (data type, sample time, etc.)
6. Depending on the results of 5, co-simulation properties have to be configured (e.g. data interpolation for different sample times, etc.)
7. The co-simulation is executed
8. Simulation results are collected and verified according to the given requirement, e.g. as described for the calibration iteration for the corresponding engineering method in Section 8.4.

8.6.2 Data artefacts

The involved data artefacts for this engineering model include:

- Input data
    - Requirements that should be verified by simulation models
    - Simulation model of different types (simulation tools, model solvers, time base, etc.)
    - Simulation model parameterization
    - Simulation engine and co-simulation parameterization
- Output data
    - Simulation results (e.g. virtual measurement variables)
    - Verified requirements

Beside the definition of data relation as described in the Section 8.3 and 8.4, the relations of the co-simulated models have to be defined such as illustrated by Figure 8-18.

8.6.3 Applying on the use case scenario

Regarding the use case scenario for UC3.4a co-simulation plays a role throughout the various testing phases. In the following two examples for the testing phase I (vehicle simulation) and phase II (engine test bed) are described.

8.6.3.1 Co-Simulation in Testing Phase I (Vehicle Simulation)

Figure 8-20 illustrates an appliance of co-simulation for the task of calibration regarding testing a hybrid vehicle. In the simulation phase, this hybrid vehicle is represented by an AVL Cruise simulation model. The model is calibrated by initial value and simulation is performed. The simulation results are then analysed by the calibration tool AVL Cameo as described by the calibration iteration engineering method (see Section 8.4).

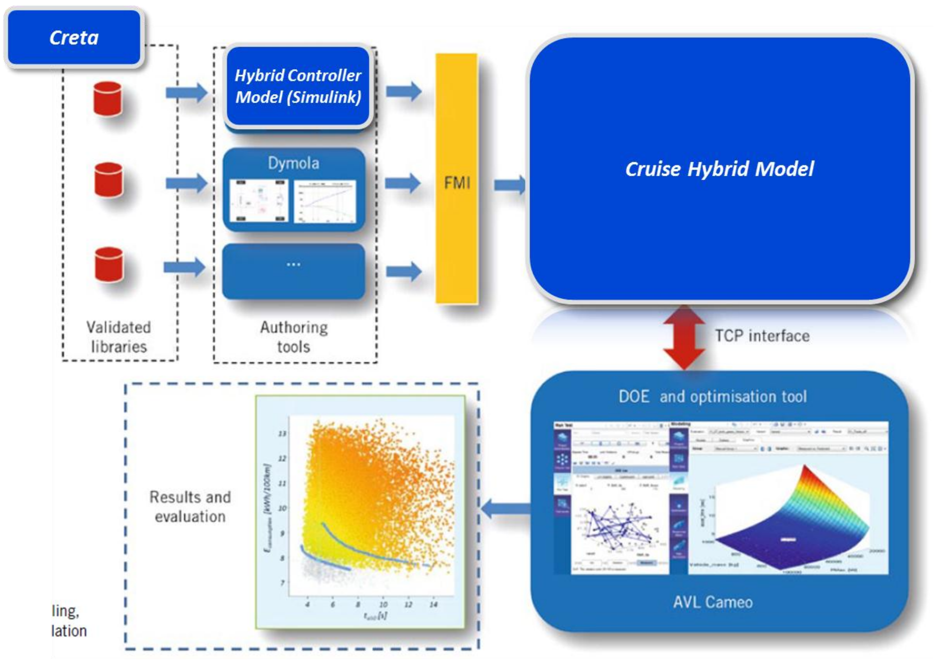


Figure 8-20 Possible interoperability scenario for co-simulation in the use case scenario

The next step in the vehicle V-model is to increase the accuracy of the simulation by a finer grained simulation of vehicle sub-components. The functionality of ECU controller is such a sub-component and Simulink models are wide-spread to model that functionality. In Figure 8-20, the Cruise hybrid model is thus extended by a hybrid controller Simulink model. The Cruise and the Simulink model should then interact at run-time via the FMI interfaces. In addition the hybrid controller model needs also to be involved in the overall calibration process. In fact, ECUs calibration is the most classical use case for calibration. Consequently, existing calibration values should be re-used from previous projects if possible. The AVL Creta tool is able to manage such calibration data and acts thus as an input source for hybrid controller model calibration as illustrated in Figure 8-20 as well.

8.6.3.2 Co-Simulation in the testing phase II (engine test bed)

Figure 8-21 illustrates the use of FMI in testing phase II: The engine test bed is controlled on a real-time operating system on the one hand, and by non-real-time configuration tools running under Windows on the other hand. Within the real-time part, different kinds of simulation models are applied, such as AVL Cruise models and Simulink models. In the latter case, Simulink models are embedded into an execution environment called ARTE.Lab. During the CRYSTAL project, co-simulation via FMI between AVL Cruise and ARTE.Lab has to be ensured.

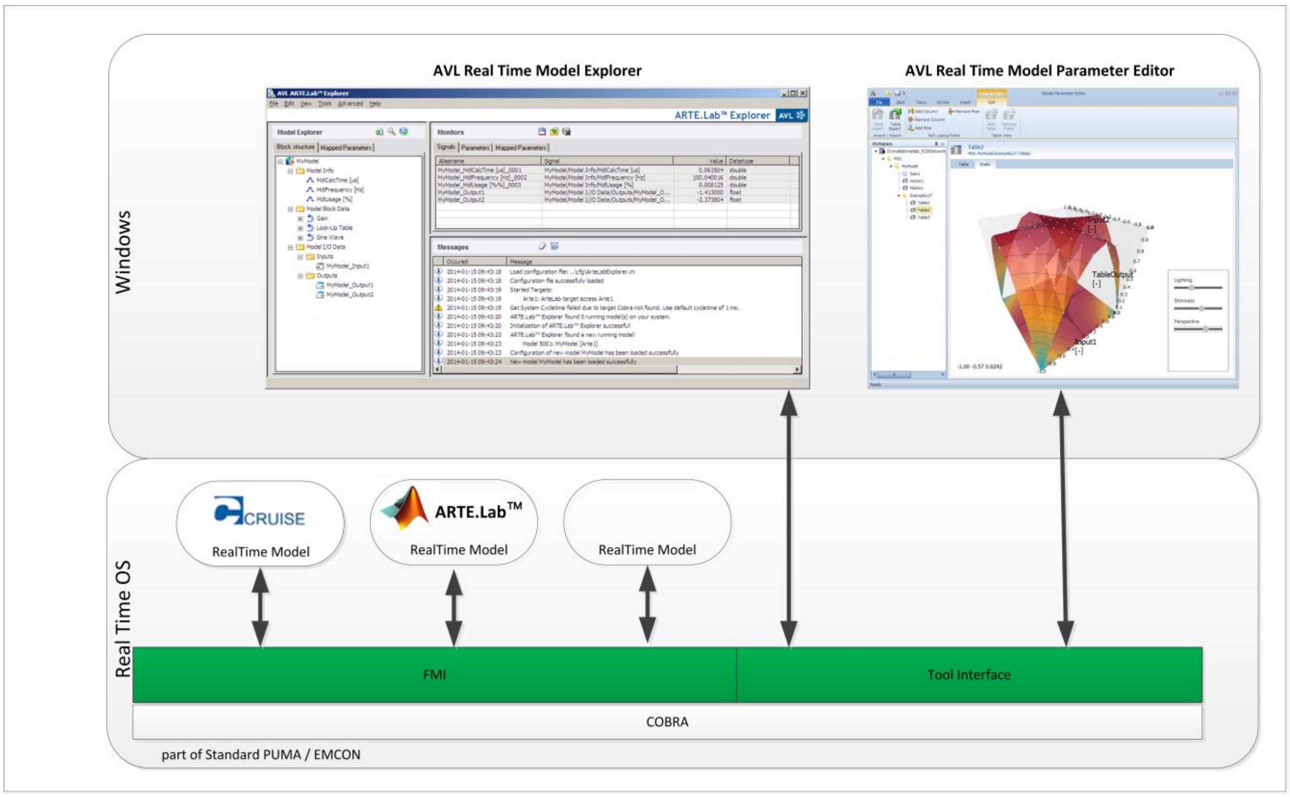


Figure 8-21 Using FMI in testing phase II

9 Identification of Engineering Methods of UC3.4b

This chapter gives a short overview about the derived engineering methods. The related activities, tools, data artefacts and their possible relation to the CRYSTAL IOS concepts are implemented in the Excel sheet of Chapter 11.1.

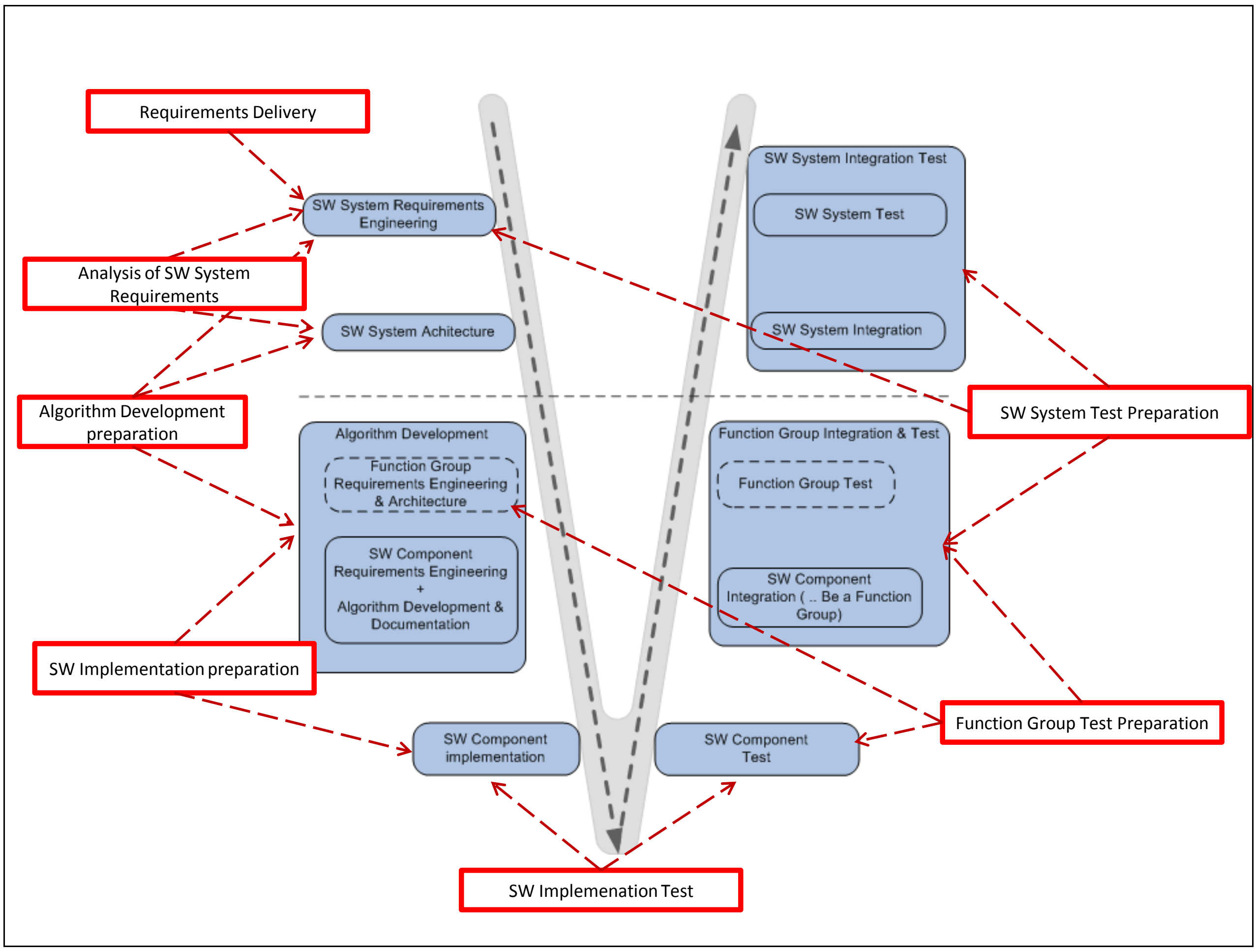

Requirements Delivery
SW System Requirements Engineering
Analysis of SW System Requirements
SW System Achitecture
Algorithm Development preparation
Algorithm Development
Function Group Requirements Engineering & Architecture
SW Component Requirements Engineering + Algorithm Development & Documentation
SW Implementation preparation
SW Component implementation
SW Implemenation Test
SW Component Test
Function Group Test Preparation
Function Group Integration & Test
Function Group Test
SW Component Integration (.. Be a Function Group)
SW System Test Preparation
SW System Integration Test
SW System Test
SW System Integration

Figure 9-1: AVL-PTE Engineering methods

9.1 Requirements Delivery

This Engineering method deals with the decomposition of Control System Requirements to SW System Requirements.

9.1.1 Engineering activities

To take place decomposition, the Control system Requirements must be checked about content and completeness and correctness.

After a successful check the Control System Requirements will be split into

- SW System Requirements.
- BSW Requirements

- Electronic Hardware Requirements
- Functional Sensor, Actuator, Busses and interfaces Requirements

and will be separated stored in the Requirements Development tool.

9.2 Analysis of SW System Requirements

To get a suitable SW System architecture, the present SW System requirements must first be analysed.

9.2.1 Engineering activities

The following step in used for the analysis of SW System Requirements via checking about the ontology and also the fitting to existing generic architecture specification. If needed, the SW System Requirements will be adapted to the existing generic architecture specification and stored in the Requirements Management tool Integrity.

9.3 Algorithm Development Preparation

To go in the development, the created Architecture will be the basis for the coming algorithm development.

9.3.1 Engineering activities

These engineering activities must be focused on the following two different aspects:

First, the link to the *AVL Data Backbone* has to be considered about the needed configuration for the import of a model.

The second focus is related to the *Integrated Tool Environment for embedded controls development*. For this, the SW System Architecture must be set in relation to the SW System Requirements. Furthermore the Version of SW System Requirements and SW System Requirements must be synchronized.

9.4 SW Implementation Preparation

After a successful (model-based) algorithm development, these algorithms have to be transformed into real software code.

9.4.1 Engineering activities

As described in Chapter 9.3, the relation between the *AVL Data Backbone* relation and the *Integrated Tool Environment* has to be considered as well. In this context the focus will be set on the configuration and import of simulation models. Additionally, the definition of deviation of MiL and SiL test and the needed configuration of the Embedded Coder must be worked out.

9.5 SW Implementation Test

The created software must be tested about reliability. These tests must be prepared in this engineering method. Regression tests and the impact of code changes during the SW tests are also in focus of this engineering method.

9.5.1 Engineering activities

This Engineering method is close related to the former engineering method about the integration of *AVL Data Backbone* by focussing on the configuration and import of the SW model.

With this engineering method further steps have to be developed to the prepare Test vectors for SiL-Test based on MiL-Test environment appropriately.

9.6 Function Group Test and SW System Test Preparation

Preparing the tests on function group level and – after that - on SW System level completes the presented development V-Cycle for Function and Software development.

9.6.1 Engineering activities

The configuration and import of a model from *AVL Data Backbone* may depend on the customer request described in the Requirements.

Nevertheless, the pre-conditions for the function group test must be collected and checked by the related engineering activities. These activities must also be applied on software system level and are basically the same as on function group level.

10 Terms, Abbreviations and Definitions

ASAM	Association for Standardization of Automation and Measuring Systems
ASAM ODS	ASAM Open Data Services
ECU	Electronic Control Unit
FMI	Functional Mock-up Interface
HiL	Hardware-in-the-Loop
IOS	Inter-Operability Specification
OSLC	Open Services for Lifecycle Collaboration
SiL	Software-in-the-Loop
SoS	System-of-Systems
SUT	System Under Test
UC	Use Case
UUT	Unit Under Test
WLTP	World-wide harmonized Light Duty Test Procedure
WP	Work Package
PSF	Powertrain Software Framework
FASD	Function And Software Development

Table 7: Terms, Abbreviations and Definitions

11 Annex I: Detailed Descriptions of the Engineering Methods

These are captured by the Excel templates. The Excel files will be inserted here, when this document is in the final version before it is submitted to the ARTEMIS JU

11.1 Engineering methods for WP3.4a

For a detailed description of the engineering methods, artefacts and activities see Chapter 8 in addition to the Excel Sheet summary:

Engineering Method: UC304_Formalize requirements_001					
Purpose: Deriving semi-formal, machine-processible requirements from natural language requirements					
Comments:					
Pre-Condition		**Engineering Activities (made of steps)**		**Post-Condition**	
Natural language requirements are stored in a requirement management tool (e.g. HP Quality Center, Doors, Requisite pro)		1. Reading natural language requirements from requirements store into a requirements formalization tool (e.g. boilerplate tool like DODT) 2. Requirements engineer derives semi-formal textual requirements by means of the refinement tool 3. Requirements are validated (consistency, completeness, correctness, redundancy, unambiguousness) 4. The semi-formal textual requirements are stored in the requirements management tool 5. Each semi-formal requirement gets linked to its corresponding natural language requirement 6. Requirements engineer creates requirement models based on the semi-formal requirements (e.g. SysML-models in Artisan studio) 7. Requirement models are stored in the modelling tool 8. Requirement models get linked to their corresponding semi-formal requirements		Semi-formal textual requirements (e.g. boiler plates) are stored in a requirement management tool and linked to their corresponding natural language requirements. SysML requirement models (based on specific SysML profile) are stored in the modelling tool and linked to the corresponding formal requirements.	
Notes: In UC304a, the natural language requirements correspond to the emission legislation WLTP (Worldwide harmonized Light duty Test Procedure)		Notes: Is sequence (textual formalization - modelling) in right order? Benefit/necessity of "graphical" requirement models? Are textual formalization and requirements modelling two Engineering Activities? Separate step for traceability links or separate engineering methods?		Notes:	
Artefacts Required as inputs of the Activities		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities**	
Name	Original requirements	Name	Boilerplate templates	Name	Formal requirements
Generic Type: (Tool or language independend type)	Natural language requirements	Type:	Text elements	Generic Type: (Tool or language independend type)	Formal textual requirement
Required Properties: (Information required in interactions between steps)	Requirement ID, Requirement statement in natural language (containing numeric limits at least as text)	Properties:	Collection of allowed text components incl. placeholders for variables, needed for composing requirement statements	Provided Properties: (Information provided in interactions between steps)	Requirement ID, Requirement statement as formal text (boilerplates), Numerical limits (minima and/or maxima) as isolated numerical attributes, Physical unit of limits
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name		Name	Ontology database	Name	
Generic Type: (Tool or language independend type)		Type:	Collection of text elements	Generic Type: (Tool or language independend type)	Requirement models in SysML
Required Properties: (Information required in interactions between steps)		Properties:	Allowed (domain-specific) artefact names	Provided Properties: (Information provided in interactions between steps)	tbd
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name		Name		Name	
Generic Type: (Tool or language independend type)		Type:		Generic Type: (Tool or language independend type)	
Required Properties: (Information required in interactions between steps)		Properties:		Provided Properties: (Information provided in interactions between steps)	
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	

Engineering Method: UC304_ValidateRequirements_001					
Purpose: Validate *Requirements* based on *Test Results* of *Test Case* (which are linked to *Requirements)*					
Comments:					
Pre-Condition		**Engineering Activities (made of steps)**		**Post-Condition**	
Semi-formal *requirements* are stored in a requirements management system (e.g. HP Quality Center, Doors, Requisite pro) *Test Cases* (that are linked to the *Requirements)* have been executed and SUT output data (e.g. measured and/or simulated time-based data channels) are stored in a database (e.g. ASAM-ODS) or in a file system		1. Requirements are read from the requirements store into a validation tool (e.g. AVL VEVAT) 2. The linked test cases and SUT output data are read 3. The validation tool analyzes the SUT output data and compares them to the requirement limits 4. Validation results (PASSED/FAILED states for all checked requirements) are generated - optionally additional information (failure reason, detected deviation from limits etc.) may be generated 5. Validation results are stored (added to the requirements) in the requirement management tool (requirements store)		For each test execution there is an individual validation result. Validation results are linked to Test Cases and stored in an appropriate data store	
		Notes: Is sequence (textual formalization - modelling) in right order? Benefit/necessity of "graphical" requirement models? Are textual formalization and requirements modelling two Engineering Activities?		Notes: Validation results usually are test-specific (i.e. depending on test input parameters the same requirement may be once PASSED or once FAILED A requirement is stated as PASSED, if every linked Test Case has been stated as PASSED	
Artefacts Required as inputs of the Activities		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities**	
Name	Formal requirements	Name	Validation rules		Validated requirements
Generic Type: (Tool or language independend type)	Semi-formal Requirements	Type:	proprietary	Generic Type: (Tool or language independend type)	semi-formal Requirements
Required Properties: (Information required in interactions between steps)	Requirement ID, Numerical limits (minima and/or maxima), Physical unit of limits, Requirement statement as formal text (boilerplates)	Properties:	Formulas, scripts, algorithm calls for analyzing the SUT output data and comparing them to requirements	Provided Properties: (Information provided in interactions between steps)	Same properties as unchecked requirements but additionally: PASSED/FAILED status and optionally further details like failure reason, detected deviation from limits etc.
Description & Interoperability Additional constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	SUT Output data	Name		Name	
Generic Type: (Tool or language independend type)	Time-based numerical data channels	Type:		Generic Type: (Tool or language independend type)	
Required Properties: (Information required in interactions between steps)	Channel names, units, sequence of numeric values etc. - as usual for measured data	Properties:		Provided Properties: (Information provided in interactions between steps)	
Description & Interoperability Additional Constraints: Standardized data file format (e.g. ASAM-ATF) or data base (e.g. ASAM ODS).		Description:		Description & Interoperability Additional Constraints:	
Name		Name		Name	
Generic Type: (Tool or language independend type)		Type:		Generic Type: (Tool or language independend type)	
Required Properties: (Information required in interactions between steps)		Properties:		Provided Properties: (Information provided in interactions between steps)	
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	

Engineering Method: UC304_TestAndCallibrationIteration_001					
Purpose: Within the overall use case, several development phases about testing a vehicle are defined. Each developent phase consists of a Test and Callibration iteration. During this iterations several data artifacts are consumed and produced by a series of tool, that need to be interconnected via IOS					
Comments:					
Pre-Condition		**Engineering Activities (made of steps)**		**Post-Condition**	
*) Requirements are well defined and formalized. *) Linking between requirements and models has been established properly. *) Access to data of previous projects and development phases has been established		1. Selection of the appropriate requirements that need to be fulfilled (see Section 8.1) 2. Selection of the appropriate simulation models that are linked to that requirements (see Section 8.3) 3. Configuration of the corresponding Calibration Set-up: a. Selection of calibration variables b. Selection of initial calibration values (e.g. from calibration models linked to similar requirements of previous projects) 4. Configuration of the corresponding test bed set-up (this task activity can be skipped in the testing phase I, since the whole test-run is based on simulation models): a. Selecting of existing test bed set-ups based on similar projects and requirements b. Creation of a new variant for this test bed setup c. Adaptation of the new variant according to the given requirements 5. Configuration of the corresponding UUT Set-up a. Configuration of the physical part of the UUT Set up such as UUT parameters (this task activity can be skipped in testing phase I) b. Configuration of the simulated part of the UUT such as simulation model parameters (rest-vehicle simulation) i. Creation of a new variant of existing simulation models of previous projects with similar requirements or simulation models used in previous testing phases. ii. Adaptation of the new created model variant according to the		*) UUT confirmes requirements *) Data artifacts are stored, exported to appropriate formats/standards and linked against each other accordingly --> It has to be reproduceable, which test results are produced with which configuration	
Notes:		Notes:		Notes:	
Artefacts Required as inputs of the Activities		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities**	
Name	Formalized requirements	Name		Name	Validation Results
Generic Type: (Tool or language independend type)	Formalized requirements	Type:		Generic Type: (Tool or language independend type)	Pair of requirement and validation result
Required Properties: (Information required in interactions between steps)	Concrete values needed for validation	Properties:		Provided Properties: (Information provided in interactions between steps)	Tells if a requirement is fullfilled by a certain test run with a given configuration or not
Description & Interoperability Additional Constraints: Level of formalization should be adaquate for requirement verification after performing a test run and the appropriate test		Description:		Description & Interoperability Additional Constraints:	
Name	Test-Cases	Name		Name	Measurement Results
Generic Type: (Tool or language independend type)	Series of input parameters for test run	Type:		Generic Type: (Tool or language independend type)	Set of values produced by a test run and an appropriate test environment (simulation models, testbed)
Required Properties: (Information required in interactions between steps)	Values in order to performe a concrete test case via simulation and/or via a test bed set-up	Properties:		Provided Properties: (Information provided in interactions between steps)	Concrete output values of the UUT for the given Test Case
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	Simulation Models	Name		Name	Applied Simulation Models
Generic Type: (Tool or language independend type)	Simulation Models for various aspects of a vehicle	Type:		Generic Type: (Tool or language independend type)	Simulation Models for various aspects of a vehicle
Required Properties: (Information required in interactions between steps)	Models should be appropriate for test case and the selected part of the UUT (e.g. engine vs. engine control SW)	Properties:		Provided Properties: (Information provided in interactions between steps)	Models should be appropriate for test case and the selected part of the UUT (e.g. engine vs. engine control SW)
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints: Variants and version of the simulation models are created during the calibration/test iteration. These have to be stored and	
Name	Testbed Configurations	Name		Name	Applied Testbed Configurations
Generic Type: (Tool or language independend type)	Series of parameter, scripts, etc. that set-up everything beside a UUT in a testbed environment	Type:		Generic Type: (Tool or language independend type)	Series of parameter, scripts, etc. that set-up everything beside a UUT in a testbed environment
Required Properties: (Information required in interactions between steps)	Configuration of measurement devices, physical dynos, fuel conditioning units etc.	Properties:		Provided Properties: (Information provided in interactions between steps)	Configuration of measurement devices, physical dynos, fuel conditioning units etc.
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints: Variants and versions of the applied testbed configuration are created during the calibration/test iteration. These have to be	
Name	Initial Calibration Values	Name		Name	Calibration Data
Generic Type: (Tool or language independend type)	Set of relevant parameter/value pairs	Type:		Generic Type: (Tool or language independend type)	Set of relevant parameter/value pairs
Required Properties: (Information required in interactions between steps)	Concrete initial values for a first educated guess of the calibration iteration	Properties:		Provided Properties: (Information provided in interactions between steps)	Final calibration data that fulfills the requirements with a specific UUT
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints: Calibration data has to be interlinked with artifacts and may act as a basis for initial calibration values for other calibration	

Engineering Method: UC304_DataManagementAcrossDevPhases_001		
Comments: Global Data Management is needed to re-use and interlink data artefacts across the several development and test phases about building and testing a vehicle		
Pre-Condition	**Engineering Activities (made of steps)**	**Post-Condition**
Interlinked artifacts (c.f. UC304_TestAndCalIteration) are available from previous AVL-wide projects or specific sub-projects and are the basis for data re-use for new testing projects. In addition there is a common infrastructure to access and store these data artifacts.	Let's assume that a customer wants to build a new vehicle. Then the engineering methods are usually comprised of the following activities: 1. Open/Create a new project for the new vehicle. This project should be defined globally and independent of the development/testing phase and the project team 2. Open a new Sub-Project for the specific development/testing phase usually assigned to a specific project team 3. Collect and define requirements for the new vehicle (eventually based on former requirements of similar projects) 4. Based in these requirements perform a pure office simulation a. For that find an initial simulation model from a similar project associated with similar requirements b. Create a new variant of this simulation model c. Refine the initial model considering overall project requirements. 5. Perform the simulation and calibration iteration as described in UC304_TestAndCalProcess. If data re-use from a previous testing phase (e.g. vehicle simulation) should be applied, the following activities are performed: 1. Based on the assumption that the development of the same vehicle started in the previous testing phase I should be continued, the associated overall vehicle project is reactivated. 2. A new sub-project associated with the actual testing phase is	The data artifacts of the finalized sub-project is accessable by the next development-phase-specific sub-project or projects for data-reuse. By integrating the engineering method such as UC304_TestAndCalibrationIteration, the data artifacts relations of a specific sub-projects are available for navigation, impact analysis, etc.
Notes:	Notes: For a proposed input and output data see the suggested data model in the Deliverably CRYSTAL_D_304_011_v1-0.doc (Section 8.5.2)	Notes:

Engineering Method: UC304_HetereogenousSimulation_001					
Purpose:					
Comments:					
Pre-Condition		**Engineering Activities (made of steps)**		**Post-Condition**	
The safety data is stored in a safety in-house tool Dysfunctional models are available		1. Based on the given requirements, the most adequate simulation models are selected from a model library as described by the engineering method in Section 8.3 (e.g. data backbone, models that are linked to similar requirements, etc.) a. If available, the models need to be modified or configured according the given requirements b. If not available, the appropriate simulation tool has to be selected c. If the selected models are based on the same simulation tool, no co-simulation is needed and thus the next steps need not to be considered 2. The selected simulation models have to be analysed regarding their type and relation to each other 3. The model relations need to be documented, e.g. by high level OSLC link relations. 4. Communication interfaces (data ports) according to the principles of FMI have to be defined. 5. Data ports have to be analysed regarding their properties (data type, sample time, etc.) 6. Depending on the results of 5, co-simulation properties have to be configured (e.g. data interpolation for different sample times, etc.) 7. The co-simulation is executed 8. Simulation results are collected and verified according to the given requirement, e.g. as described for the calibration iteration for the corresponding engineering method		Fault-trees are generated	
Notes:		Notes:		Notes:	
Artefacts Required as inputs of the Activities		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities**	
Name	Requirements			Name	Simulation results
Generic Type: (Tool or language independend type)	Requirement			Generic Type: (Tool or language independend type)	Data sequence of (measurement) values
Required Properties: (Information required in interactions between steps)	Formal or informal representation			Provided Properties: (Information provided in interactions between steps)	Type of value, physical dimension, value itself
Description & Interoperability Additional Constraints: Requirements should be verified by simulation models				Description & Interoperability Additional Constraints:	
Name	Simulation model			Name	
Generic Type: (Tool or language independend type)	Model			Generic Type: (Tool or language independend type)	
Required Properties: (Information required in interactions between steps)	Model Element, Sub-Element, Model-Parameter			Provided Properties: (Information provided in interactions between steps)	
Description & Interoperability Additional Constraints: Simulation model of different types should be supported (simulation tools, model solvers, time base, etc.)				Description & Interoperability Additional Constraints:	
Name	Simulation model parameterization				
Generic Type: (Tool or language independend type)	Parameter				
Required Properties: (Information required in interactions between steps)	Parameter Name, Parameter Value				
Description & Interoperability Additional Constraints:					

11.2 Engineering methods for WP3.4b

Engineering Method: UC304b_RequirementsDelivery_001		
Purpose: Delivery of SW System Requirements		
Comments:		
Pre-Condition	**Engineering Activities (made of steps)**	**Post-Condition**
2. Integrated Tool Environment for embedded controls development Natural language requirements are stored in a Requirement Management tool (Integrity). 3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to required artefacts as input for the activites	2. Integrated Tool Environment for embedded controls development Decomposition of Control System Requirements to SW System Requirements 1) Check of Control system Requirements 2) Splitting of Control System Requirements into - SW System Requirements. - BSW Requirements - Electronic Hardware Requirements - Functional Sensor, Actuator, Busses and interfaces Requirements 3) Store the SW System Requirements in the Requirements Development tool. 3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to internal used artefacts within the activites	2. Integrated Tool Environment for embedded controls development Basis for SW System Requirements stored in Requirements management Tool (Integrity). 3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to provided artefacts as output for the activites.
Notes: The here mentioned "Natural language requirements" are "Control System Requirements" (Level3 Requirements). Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS specification.	Notes: The Control System Requirements (Level 4 Requirements) could be delivered from Customer or from CoSyp (previous development process).	Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS specification.

Artefacts Required as inputs of the Activities to Integrated Tool Environment for embedded controls development		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities** Integrated Tool Environment for embedded controls development	
Name	Control System Requirements	Name		Name	Basis SW System Requirements
Generic Type: (Tool or language independend type)	Natural Language Requirement	Type:		Generic Type: (Tool or language independend type)	Natural Language Requirement
Required Properties: (Information required in interactions between steps)	ID, System Level, Category, Requirement Text, Requirement Summary,Rationale, Acceptence Criteria, Stakeholder Comment, State, Priority, Responsibility, Level=4, Origin, Safety Level, Safety Standard, Text Attachments, Re-Use Potential, System Structure Nodes, Stakeholder ID	Properties:		Provided Properties: (Information provided in interactions between steps)	ID, System Level, Category, Requirement Text, Requirement Summary,Rationale, Acceptence Criteria, Stakeholder Comment, State, Priority, Responsibility, Level=5, Origin, Safety Level, Safety Standard, Text Attachments, Re-Use Potential, System Structure Nodes, Stakeholder ID
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	

Engineering Method: UC304b_AnalysisOfSwSystemRequirements_002		
Purpose: Check about the quality of a requirements for further usage in SW System Architecture definition		
Comments:		
Pre-Condition	**Engineering Activities (made of steps)**	**Post-Condition**
2. Integrated Tool Environment for embedded controls development SW System Requirements stored in Requirements management Tool (Integrity).	2. Integrated Tool Environment for embedded controls development Analysis of SW System Requirements 1) Checked about onthology 2) Check if the SW System Requirements are fitting to existing generic architecture specification 3) Adaptation of SW System Requirements to existing generic architecture specification. 4) Store the SW System Requirements in the Requirements Managment tool (Integrity).	2. Integrated Tool Environment for embedded controls development Approved or modified SW System Requirements stored in Integrity.
3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to required artefacts as input for the activites	3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to internal used artefacts within the activites	3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to provided artefacts as output for the activites.
Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS	Notes: Generic architecture specification is stored in Integrity-source	Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS

Artefacts Required as inputs of the Activities to Integrated Tool Environment for embedded controls development		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities** Integrated Tool Environment for embedded controls development	
Name	SW System Requirements	Name		Name	SW System Requirements
Generic Type: (Tool or language independend type)	Natural Language Requirement	Type:		Generic Type: (Tool or language independend type)	Natural Language Requirement
Required Properties: (Information required in interactions between steps)	ID, System Level, Category, Requirement Text, Requirement Summary,Rationale, Acceptence Criteria, Stakeholder Comment, State, Priority, Responsibility, Level=5, Origin, Safety Level, Safety Standard, Text Attachments, Re-Use Potential, System Structure Nodes, Stakeholder ID	Properties:		Provided Properties: (Information provided in interactions between steps)	ID, System Level, Category, Requirement Text, Requirement Summary,Rationale, Acceptence Criteria, Stakeholder Comment, State, Priority, Responsibility, Level, Origin, Safety Level, Safety Standard, Text Attachments, Re-Use Potential, System Structure Nodes, Stakeholder ID
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	

Engineering Method: UC304b_AlgorithmDevelopmentPreparation_003 - really needed?		
Purpose: Preparation of Algorithm development		
Comments:		
Pre-Condition	**Engineering Activities (made of steps)**	**Post-Condition**
1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform Configuration data to be used Model from Data Backbone 2. Integrated Tool Environment for embedded controls development - SW system requirements stored in Integrity - SW System Architecture stored Architecture Tool. 3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to required artefacts as input for the activites	1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform a) Configure and import Model from AVL Data Backbone b) Import of Model from Data Backbone 2. Integrated Tool Environment for embedded controls development a) Set SW System Architecture in relation to the SW System Requirements. b) Version of SW System Requirements and SW System Requirements must be synchronized c) Fix Versions in Intergrity 3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to internal used artefacts within the activites	1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform Baselined model from AVL Data Backbone to be used in Algorithm development 2. Integrated Tool Environment for embedded controls development Baselines of SW System Requirements and SW Architecture within Integrity. 3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to provided artefacts as output for the activites.
Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS specification.	Notes: … to be compatible with to existing Architecture rules. … to be compatible with to existing Architecture rules.	Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS specification.

Artefacts Required as inputs of the Activities AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities** AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform	
Name	Configuration Data to be used Model from Data Backbone	Name		Name	Baselined model from AVL Data Backbone to be used in
Generic Type: (Tool or language independend type)	Configuration data	Type:		Generic Type: (Tool or language independend type)	Model
Required Properties: (Information required in interactions between steps)	Parameter depend on the the model configuration. A List of Requirements is to long.	Properties:		Provided Properties: (Information provided in interactions between steps)	Compiled Model to be used in Target System. Compiled S-Function for Matlab Simulink System
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	

Artefacts Required as inputs of the Activities to Integrated Tool Environment for embedded controls development		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities** Integrated Tool Environment for embedded controls development	
Name	SW System Requirements	Name		Name	Baselines of SW System Requirements
Generic Type: (Tool or language independend type)	Natural Language Requirement	Type:		Generic Type: (Tool or language independend type)	Snapshot of the SW System Requirements
Provided Properties: (Information provided in interactions between steps)	ID, System Level, Category, Requirement Text, Requirement Summary,Rationale, Acceptence Criteria, Stakeholder Comment, State, Priority, Responsibility, Level=5, Origin, Safety Level, Safety Standard, Text Attachments, Re-Use Potential, System Structure Nodes, Stakeholder ID	Properties:		Provided Properties: (Information provided in interactions between steps)	
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	SW System Architecture	Name		Name	Baselines of SW Architecture
Generic Type: (Tool or language independend type)	Stored in Architecture Tool	Type:		Generic Type: (Tool or language independend type)	Snapshot of the SW System Architecture
Provided Properties: (Information provided in interactions between steps)	Architecture of the System	Properties:		Provided Properties: (Information provided in interactions between steps)	
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	

Engineering Method: UC304b_SwImplementationPreparation_004					
Purpose: Preparation of the Software implementation					
Comments:					
Pre-Condition		**Engineering Activities (made of steps)**		**Post-Condition**	
1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform Configuration data to be used Model from Data Backbone		1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform a) Configure and import Model from AVL Data Backbone b) Import of Model from Data Backbone Remark: Depend on the "usage" mechanism of the model from the AVL Data back bone, a) and b) could be obsolete and the model from the former Engineering method can be used with adaptation to this activity.		1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform Baselined model from AVL Data Backbone to be used in Algorithm development	
2. Integrated Tool Environment for embedded controls development - Software Component Requirements (for Function Group / Function Component) stored in Integrity - SW Component Model stored in Integrity-Source - SW Component Documentation stored in Integrity-Source. - Architecture (Interfaces) on Function Group Level		2. Integrated Tool Environment for embedded controls development - defintion of deviation of MiL and SiL test ist allowed - definition of Embedded Coder configuration		2. Integrated Tool Environment for embedded controls development Embedded Coder configuration	
3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to required artefacts as input for the activites		3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to internal used artefacts within the activites		3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to provided artefacts as output for the activites.	
Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS		Notes: for Autocode creation for floating point model		Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS	
Artefacts Required as inputs of the Activities AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities** AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform	
Name	Configuration Data to be used Model from Data Backbone	Name		Name	Baselined model from AVL Data Backbone to be used in
Generic Type: (Tool or language independend type)	Configuration data	Type:		Generic Type: (Tool or language independend type)	Model
Required Properties: (Information required in interactions between steps)	Parameter depend on the the model configuration. A List of Requirements is to long.	Properties:		Provided Properties: (Information provided in interactions between steps)	Compiled Model to be used in Target System. Compiled S-Function for Software System
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Artefacts Required as inputs of the Activities to Integrated Tool Environment for embedded controls development		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities** Integrated Tool Environment for embedded controls development	
Name	SW Component Requirements	Name		Name	
Generic Type: (Tool or language independend type)	ID, System Level, Category, Requirement Text, Requirement Summary,Rationale, Acceptence Criteria, Stakeholder Comment, State, Priority, Responsibility, Level=5, Origin, Safety Level, Safety Standard, Text Attachments, Re-Use Potential, System Structure Nodes, Stakeholder ID	Type:		Generic Type: (Tool or language independend type)	
Provided Properties: (Information provided in interactions between steps)		Properties:		Provided Properties: (Information provided in interactions between steps)	
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	SW Component Model	Name		Name	Embedded Coder configuration
Generic Type: (Tool or language independend type)	Simulink, ASCET, ...	Type:		Generic Type: (Tool or language independend type)	GNU
Provided Properties: (Information provided in interactions between steps)	*.m -File *.ddx File	Properties:		Provided Properties: (Information provided in interactions between steps)	Configuration File with configuration parameters
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	SW Component Documentation	Name		Name	
Generic Type: (Tool or language independend type)	Natural Language Documentation	Type:		Generic Type: (Tool or language independend type)	
Provided Properties: (Information provided in interactions between steps)		Properties:		Provided Properties: (Information provided in interactions between steps)	
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	Function Group Architecture	Name		Name	
Generic Type: (Tool or language independend type)	Stored in Architecture Tool	Type:		Generic Type: (Tool or language independend type)	
Provided Properties: (Information provided in interactions between steps)	Function Group Architecture of the System	Properties:		Provided Properties: (Information provided in interactions between steps)	
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	

Engineering Method: UC304b_SwImplementationTest_005		
Purpose: Test of the SW implementation		
Comments:		
Pre-Condition	**Engineering Activities (made of steps)**	**Post-Condition**
1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform Configuration data to be used Model from Data Backbone	1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform a) Configure and import Model from AVL Data Backbone b) Import of Model from Data Backbone Remark: Depend on the "usage" mechanism of the model from the AVL Data back bone, a) and b) could be obsolete and the model from the former Engineering method can be used with adaptation to this activity.	1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform Baselined model from AVL Data Backbone to be used in Algorithm development
2. Integrated Tool Environment for embedded controls development - Software Component Code (for Function Group / Function Component) stored in Integrity - SW Component Model stored in Integrity-Source - Result of MiL-Test from algorithm development stored in Integrity-Source	2. Integrated Tool Environment for embedded controls development Preparation of Test vectors for SiL-Test based on MiL-Test environment. ... adaptation of Testvectors for SiL-Test	2. Integrated Tool Environment for embedded controls development Test vectors for SiL-Test. C-Code based on MiL-Test model.
3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to required artefacts as input for the activites	3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to internal used artefacts within the activites	3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to provided artefacts as output for the activites.
Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS	Notes: for Autocode creation for floating point model. Depend on the "usage" mechanism of the model from the AVL	Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS

Artefacts Required as inputs of the Activities AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities** AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform	
Name	Configuration Data to be used Model from Data Backbone	Name		Name	Baselined model from AVL Data Backbone to be used in
Generic Type: (Tool or language independend type)	Configuration data	Type:		Generic Type: (Tool or language independend type)	Model
Required Properties: (Information required in interactions between steps)	Parameter depend on the the model configuration. A List of Requirements is to long.	Properties:		Provided Properties: (Information provided in interactions between steps)	Compiled Model to be used in Target System. Compiled S-Function for Software System
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	

Artefacts Required as inputs of the Activities to Integrated Tool Environment for embedded controls development		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities** Integrated Tool Environment for embedded controls development	
Name	Software Component Code	Name		Name	Test vector
Generic Type: (Tool or language independend type)	C++	Type:		Generic Type: (Tool or language independend type)	Test suite stored in Integrity
Required Properties: (Information required in interactions between steps)	code of the Software Compoenent with interfaces to other components	Properties:		Provided Properties: (Information provided in interactions between steps)	to be final defined (due to current implementation in Integrity
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	SW Component Model	Name		Name	C-Code
Generic Type: (Tool or language independend type)	Matlab Simulink, ASCET	Type:		Generic Type: (Tool or language independend type)	C++
Required Properties: (Information required in interactions between steps)	Model of the Software Compoenent with interfaces to other components	Properties:		Provided Properties: (Information provided in interactions between steps)	code of the Software Compoenent with interfaces to other components
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	Result of MiL-Test from algorithm development	Name		Name	
Generic Type: (Tool or language independend type)		Type:		Generic Type: (Tool or language independend type)	
Required Properties: (Information required in interactions between steps)		Properties:		Provided Properties: (Information provided in interactions between steps)	
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	

Engineering Method: UC304b_FGTestPreparation_006					
Purpose: Preparation of the Function Group Tests					
Comments:					
Pre-Condition		**Engineering Activities (made of steps)**		**Post-Condition**	
1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform Configuration data to be used Model from Data Backbone		1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform a) Configure and import Model from AVL Data Backbone b) Import of Model from Data Backbone Remark: Depend on the "usage" mechanism of the model from the AVL Data back bone, a) and b) could be obsolete and the model from the former Engineering method can be used with adaptation to this activity.		1. AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform Baselined model from AVL Data Backbone to be used in Algorithm development	
2. Integrated Tool Environment for embedded controls development - SW components (including Test and Review) - FG-Architecture - FG requirements - FG test cases		2. Integrated Tool Environment for embedded controls development Collection, check of all Pre-conditions		2. Integrated Tool Environment for embedded controls development - SW components (including Test and Review) - FG-Architecture - FG requirements - FG test cases	
3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to required artefacts as input for the activites		3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to internal used artefacts within the activites		3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to provided artefacts as output for the activites.	
Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS		Notes:		Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS	
Artefacts Required as inputs of the Activities AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities** AVL Data Backbone to enable models exchange with integration and configuration of versatile System-of-systems (SoS) platform	
Name	Configuration Data to be used Model from Data Backbone	Name		Name	Baselined model from AVL Data Backbone to be used in
Generic Type: (Tool or language independend type)	Configuration data	Type:		Generic Type: (Tool or language independend type)	Model
Required Properties: (Information required in interactions between steps)	Parameter depend on the the model configuration. A List of Requirements is to long.	Properties:		Provided Properties: (Information provided in interactions between steps)	Compiled Model to be used in Target System. Compiled S-Function for Software System
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Artefacts Required as inputs of the Activities to Integrated Tool Environment for embedded controls development		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities** Integrated Tool Environment for embedded controls development	
Name	Software Component Code	Name		Name	Software Component Code
Generic Type: (Tool or language independend type)	C++	Type:		Generic Type: (Tool or language independend type)	C++
Required Properties: (Information required in interactions between steps)	code of the Software Compoenent with interfaces to other components	Properties:		Provided Properties: (Information provided in interactions between steps)	code of the Software Compoenent with interfaces to other components
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	Function Group Architecture	Name		Name	Function Group Architecture
Generic Type: (Tool or language independend type)	Structure stored in Integrtiy	Type:		Generic Type: (Tool or language independend type)	Structure stored in Integrtiy
Required Properties: (Information required in interactions between steps)	Architecture of Function Group	Properties:		Provided Properties: (Information provided in interactions between steps)	Architecture of Function Group
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	Function Group Requirements	Name		Name	Function Group Requirements
Generic Type: (Tool or language independend type)	Natural Language Requirement	Type:		Generic Type: (Tool or language independend type)	Natural Language Requirement
Required Properties: (Information required in interactions between steps)	ID, System Level, Category, Requirement Text, Requirement Summary,Rationale, Acceptence Criteria, Stakeholder Comment, State, Priority, Responsibility, Level=5, Origin, Safety Level, Safety Standard, Text Attachments, Re-Use Potential, System Structure Nodes, Stakeholder ID	Properties:		Provided Properties: (Information provided in interactions between steps)	ID, System Level, Category, Requirement Text, Requirement Summary,Rationale, Acceptence Criteria, Stakeholder Comment, State, Priority, Responsibility, Level=5, Origin, Safety Level, Safety Standard, Text Attachments, Re-Use Potential, System Structure Nodes, Stakeholder ID
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	Function Group test cases	Name		Name	Function Group test cases
Generic Type: (Tool or language independend type)	Test cases stored in Integrity	Type:		Generic Type: (Tool or language independend type)	Test cases stored in Integrity
Required Properties: (Information required in interactions between steps)	to be final defined (due to current implementation in Integrity	Properties:		Provided Properties: (Information provided in interactions between steps)	to be final defined (due to current implementation in Integrity
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	

Engineering Method: UC304b_SwSystemTestPreparation_007					
Purpose: Preparation of the SW System Tests					
Comments:					
Pre-Condition		**Engineering Activities (made of steps)**		**Post-Condition**	
2. Integrated Tool Environment for embedded controls development - All Function Groups (including Test and Review) - SW System Architecture - SW System requirements - SW System test cases 3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to required artefacts as input for the activites		2. Integrated Tool Environment for embedded controls development Collection, check of all informations 3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to internal used artefacts within the activites		2. Integrated Tool Environment for embedded controls development - All Function Groups (including Test and Review) - SW System Architecture - SW System requirements - SW System test cases 3. Improvement of the Development via „Efficient Variant Handling within V-Cycle" Variant information linked to provided artefacts as output for the activites.	
Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS		Notes:		Notes: Variant handling cannot be considered as an interoperability challenge on its own, but has to be considered for the IOS	
Artefacts Required as inputs of the Activities to Integrated Tool Environment for embedded controls development		**Artefacts used internally within the Activities (optional)**		**Artefacts Provided as outputs of the Activities** Integrated Tool Environment for embedded controls development	
Name	All Function Groups	Name		Name	All Function Groups
Generic Type: (Tool or language independend type)	Matlab Simulink model, C-Code, Documentation, *.ddx Container, Model tests, SW tests	Type:		Generic Type: (Tool or language independend type)	Matlab Simulink model, C-Code, Documentation, *.ddx Container, Model tests, SW tests
Required Properties: (Information required in interactions between steps)	Matlab Simulink model, C-Code, Documentation, *.ddx Container, Model tests, SW tests	Properties:		Provided Properties: (Information provided in interactions between steps)	Matlab Simulink model, C-Code, Documentation, *.ddx Container, Model tests, SW tests
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	SW System Architecture	Name		Name	SW System Architecture
Generic Type: (Tool or language independend type)	Stored in Architecture Tool	Type:		Generic Type: (Tool or language independend type)	Stored in Architecture Tool
Required Properties: (Information required in interactions between steps)	Architecture of the System	Properties:		Provided Properties: (Information provided in interactions between steps)	Architecture of the System
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	SW System Requirements	Name		Name	SW System Requirements
Generic Type: (Tool or language independend type)	Natural Language Requirement	Type:		Generic Type: (Tool or language independend type)	Natural Language Requirement
Required Properties: (Information required in interactions between steps)	ID, System Level, Category, Requirement Text, Requirement Summary,Rationale, Acceptence Criteria, Stakeholder Comment, State, Priority, Responsibility, Level=5, Origin, Safety Level, Safety Standard, Text Attachments, Re-Use Potential, System Structure Nodes, Stakeholder ID	Properties:		Provided Properties: (Information provided in interactions between steps)	ID, System Level, Category, Requirement Text, Requirement Summary,Rationale, Acceptence Criteria, Stakeholder Comment, State, Priority, Responsibility, Level=5, Origin, Safety Level, Safety Standard, Text Attachments, Re-Use Potential, System Structure Nodes, Stakeholder ID
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	
Name	SW System test cases	Name		Name	SW System test cases
Generic Type: (Tool or language independend type)	Test cases stored in Integrity	Type:		Generic Type: (Tool or language independend type)	Test cases stored in Integrity
Required Properties: (Information required in interactions between steps)	to be final defined (due to current implementation in Integrity	Properties:		Provided Properties: (Information provided in interactions between steps)	to be final defined (due to current implementation in Integrity
Description & Interoperability Additional Constraints:		Description:		Description & Interoperability Additional Constraints:	