OETCS/WP4/D4.1V01.02

ITEA2 Project
Call 6 11025
2012 – 2015

Work Package 4: “Validation & Verification Strategy”

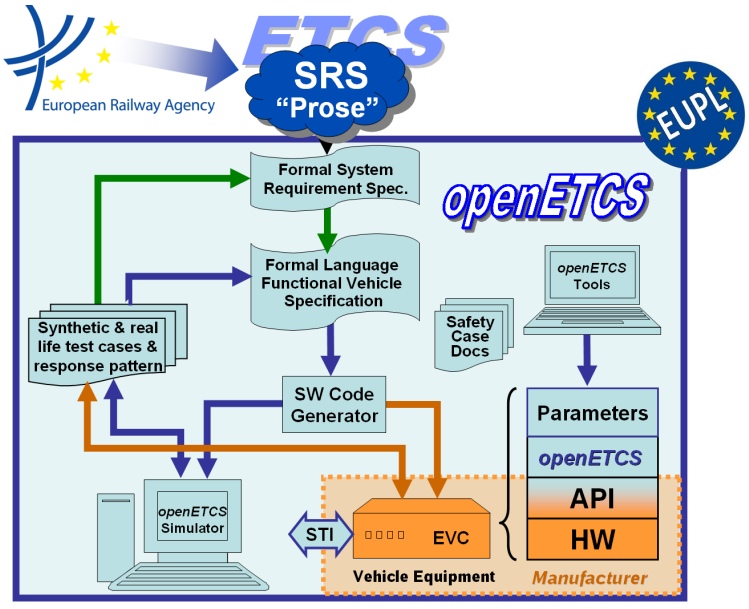
openETCS Validation & Verification Plan

Version 01.02

Hardi Hungar (Ed.)

Contributions by:
Frederic Badeau (Systerel), Marc Behrens (DLR),
Cecile Braunstein (U Bremen), Cyril Cornu (All4Tec),
Christophe Gaston (CEA), Jens Gerlach (Fraunhofer),
Ainhoa Gracia (SQS), Hardi Hungar (DLR),
Stephan Jagusch (AEbt), Alexander Nitsch (U Rostock),
Jan Peleska (U Bremen), Marielle Petit-Doche (Systerel),
Virgile Prevosto (CEA), Stefan Rieger (TWT),
Izaskun de la Torre (SQS), Jan Welte(TU-BS)

16 July, 2014



Funded by:

Federal Ministry of Education and Research
Région de Bruxelles-Capitale
GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, ENERGÍA Y TURISMO

openETCS

This page is intentionally left blank

Work Package 4: "Validation & Verification Strategy" **OETCS/WP4/D4.1V01.02**
16 July, 2014

openETCS Validation & Verification Plan

Version 01.02

Document approbation

Lead author:	Technical assessor:	Quality assessor:	Project lead:
location / date	location / date	location / date	location / date
signature Hardi Hungar (DLR)	signature Marc Behrens (DLR)	signature Jens Gerlach (Fraunhofer FOKUS)	signature Klaus-Rüdiger Hase (DB Netz)

Hardi Hungar (Ed.)
Contributions by:
Frederic Badeau (Systerel), Marc Behrens (DLR),
Cecile Braunstein (U Bremen), Cyril Cornu (All4Tec),
Christophe Gaston (CEA), Jens Gerlach (Fraunhofer),
Ainhoa Gracia (SQS), Hardi Hungar (DLR),
Stephan Jagusch (AEbt), Alexander Nitsch (U Rostock),
Jan Peleska (U Bremen), Marielle Petit-Doche (Systerel),
Virgile Prevosto (CEA), Stefan Rieger (TWT),
Izaskun de la Torre (SQS), Jan Welte(TU-BS)

DLR
Lilienthalplatz 7
38108 Brunswick, Germany
eMail:hardi.hungar@dlr.de

Deliverable

Prepared for openETCS@ITEA2 Project

Abstract: This document describes strategy and plan of the verification and validation in the project openETCS. It revises the previous version (V01) of this document.

The overall goal of openETCS is to develop the software of the EVC. It starts from the specification given in Subset 026, formalizes this, provides a software-hardware interface and ends with a software for the EVC. This development is to be done in a FLOSS style, as far as possible. This goal will only partly be achieved within the current ITEA 2 *project*.

Following some introductory chapters, ths document has three main parts. The first defines verification and validation for the full development. This part shall eveolve into a draft of a verification and validation plan for *activities* which take up the results of the current ITEA 2 project. The second part plans only those activities which will actually be performed within the current project. These will be related to the overall plan in the first part, and they will be described in greater detail.

Both parts also address the issue of tools to support verification & validation activities, which is another concern of openETCS. Details of tools and methods, and how they could contribute to achieve the goals of openETCS, are given in the third part of the document.

It is planned to further revise this document (V03) after the currently planned round of verification & validation activities the second level, and to use it as a basis for deliverable D4.4, the Final Report on Verification & Validation.

Some of the revision work needed to be done for this version is indicated in the form of comments like this one. Most of the comments are missing themselves, though.

Table of Contents

Figures and Tables

Figures

Tables

Document Control

Document information	
Work Package	WP4
Deliverable ID or doc. ref.	D4.1
Document title	openETCS Validation & Verification Plan
Document version	01.02
Document authors (org.)	Marc Behrens (DLR), Cecile Braunstein (U Bremen), Cyril Cornu (All4Tec), Christophe Gaston (CEA), Jens Gerlach (Fraunhofer), Ainhoa Gracia (SQS), Hardi Hungar (DLR), Stephan Jagusch (AEbt), Alexander Nitsch (U Rostock), Jan Peleska (U Bremen), Virgile Prevosto (CEA), Stefan Rieger (TWT), Izaskun de la Torre (SQS), Frederic Badeau (Systerel), Marielle Petit-Doche (Systerel)

Review information	
Last version reviewed	–
Main reviewers	–

Approbation			
	Name	Role	Date
Written by	Hardi Hungar	WP4-T4.1 Task Leader	May 2014
Approved by	Marc Behrens	WP4 Leader	*tbd*

Document evolution			
Version	Date	Author(s)	Comment
01.01	22/05/2014	H. Hungar	Revision of document structure and content (partially) based on V01.00 and preliminary versions of the reports D4.2
01.02	16/07/2014	H. Hungar	Added DLR VnV User Story
01.03			
02.00	dd.mm.2014	M. Behrens	Review and approval

1 Introduction

1.1 Purpose and Structure of the Document

This document describes strategy and plan of the verification and validation in openETCS. It revises the previous version (V01) of this document.

Perhaps use parts of / refer to the new version of the FPP.

We distinguish here between the current openETCS project, funded by ITEA, and the openETCS activity as a whole, which encompasses the project.

- The openETCS *activitiy* pursues the vision of a full FLOSS development of the software of the *European Vital Computer* (EVC). That starts from the specification given in Subset 026 with a formalisation of this specification. Then, a software-hardware interface is defined and the requirements on the software are derived. The final software shall come with a development documentation which enables manufacturers to take up the result and use it for the construction of their products. This entails that the whole approach shall be SIL 4 compliant. The software shall as far as possible be *Free, Libre, Open-Source Software* (FLOSS)—that is, it shall be open source, developed with freely accessible tools, including verification and validation.
- The openETCS *project* is the current activity funded by ITEA. The project is a major first step in realising the goals of the activity. It shall demonstrate that the openETCS vision can be realised, and perform some substantial steps towards that realisation. The project will define process and methods, select suitable tools and do some part of the development itself.

Accordingly, the current document addresses, in different parts, both the openETCS activity and its ongoing realisation within the project.

1. Following some introductory sections, the first part defines verification and validation for the full development. In its final version, it shall be a ¿nearly complete? CENELEC-compliant plan for verification and validation of the openETCS EVC software.
2. The second one plans only those activities which will actually be performed within the current project. These will be related to the overall plan in the first part, and they will be described in greater detail.
3. A third part collects descriptions of methods and tools. Most of these are already available from project partners or third parties. Some of them are subject to adaptations or even further development within openETCS. The first and second part refer to relevant methods and tools which are used or could be used for verification or validation, and, vice versa, the descriptions specify for which activity they can be used.

Each of the first two parts defines, in different sections, the verification & validation strategy, the verification plan and the validation plan. Verification and validation share some of their methods and tools, and in some case are applied to the same design artifacts. Therefore, the plans for both are included in this document. Nevertheless, these activities are intended to be and remain independent.

The document will refer to the Quality Assurance Plan [1] for a definition of development phases and artifact denotations. Since this is currently still under review these references will not be final and will have to be revised.

Verification and validation play an important role in the safety case. This document identifies the V&V activities which do contribute and refers to the safety plan for further details on the additional requirements to be met and a precise statement of what has to be established.

It is planned to further revise this document (V03) after the currently planned round of verification & validation activities ¿naming?, and to use it as a basis for deliverable D4.4, the Final Report on Verification & Validation. The first part shall evolve into

OLD: *There are three main issues which make this plan different from an ordinary V&V plan for a software to develop. First, openETCS is not only concerned with the software part of the EVC. As part of the activities, a semi-formal model for SS 026 is to be developed and to be verified. As the SS 026 covers parts of the ETCS system beyond the software, also process steps on the system (not just software) level are to be performed. And in particular the design does not start with a clearly defined set of requirements on the software.*

As a second point, openETCS will not only do development, but shall also be concerned with processes, methods and tools with the goal of being able to propose a complete SIL 4 compliant approach. As part of this, it has to be defined how to handle verification and validation. This is done in the sections addressing a "full development". Due to the limited resources of the project, actually performing such a full development is out of the project's scope. Instead, only some functions will be implemented, and only partial lines of development will be realised. V&V related to these activities is to be planned in the specific sections dedicated to "openETCS".
DLO

1.2 Background Information

1.2.1 Definitions

Verification

Verification is an activity which has to be performed at each step of the design. It has to be verified that the design step achieved its goals. This consists at least of two parts:

- that the artifacts produced in the step are of the right type and contain all the information they should. E.g., that the SSRS identifies all components addressed in SS 026, specifies their interfaces in sufficient detail and has allocated the functions to the components (this should just serve an example and is based on a guess what the SSRS should do)
- that the artifact correctly implements the input requirements of the design step. These typically include the main output artifacts of the previous step. "Correctly implements" includes requirement coverage (tracing). This can and should be supported by some tools.

Adequacy of such tools depends on things like format compatibility, degree of automation, functionality (e.g., ability to handle m-to-n relations). Depending on the design step (and the nature of the artifacts) different forms of verification will complement requirement coverage, with different levels of support. The step from SS 026 to the SSRS will mainly consist of manual activities besides things like coverage checks. Verifying a formal (executable) model against the SSRS can be supported by animation or simulation to e.g. execute test cases which have been designed to check compliance with the SSRS. Even formal proof tools may be employed to check or establish properties. Model-to-code steps offer far more options (and needs) for tool support. And tools or tool sets for unit test will support dynamic testing for requirement or code coverage. This may include test generation, test execution with report generation, test result evaluation and so on. Also, code generator verification (or qualification) may play a role, here. Integration steps mandate still other testing (or verification) techniques.

Summarizing, one may say that verification subsumes highly diverse activities, and may be realized in very many different forms.

Validation

Validation is name for the activity by which the compliance of the end result with the initial requirements is shown. In the case of openETCS, this means that the demonstrator (or parts of it) are checked against the SS 026 or one of its close descendants (i.e., SSRS), taking also further sources of requirements from operational scenarios and TSIs into accoutn. This will consist of testing the equipment according to a test plan derived from the requirements and detailed into concrete test cases at some later stage. Tool support for validation will thus mainly concern test execution and evaluation, perhaps supplemented by test derivation or test management. Ambitious techniques like formal proof are most likely not applicable here.

Thus, the tool support for validation will not differ substantially from that for similar verification activities.

One might also consider “early” validation activities, e.g. “validating” an executable model against requirements from the SS 026. These are not mandated by the standards and can per se not replace verification of design steps. They may nevertheless be worthwhile as means for early defect detection.

Further (mostly complementary) information on V&V can be found in the report on the CENELEC standards (D2.2).

2 Document Evolution

The verification and validation plan shall be revised in the course of the project as the design progresses and gets detailed and experiences with verification and validation are made. This is in accordance with the EN 50128, where it is required that the plan shall be maintained throughout the development cycle.

V01, T0+18: First version of the plan.

V02, T0+22: First revision (this document), based on the 1st V&V interim reports on applicability of the V&V approach to model and implementation/code (D4.2.1, D4.2.2), and a definition of the hazard and risk analysis methodology (D4.2.3)

V03, T0+30: Second revision, based on the internal reports on the applicability of the V&V approach to prototypes of design models and code

V04, T0+42: Final version as part of the final V&V report (D4.4)

OLD: *The first version of the plan was based on the available information of the design process. This is not yet very detailed as also the description in Chapter 3 of this report shows. In particular, the nature of the SSRS is yet to be defined precisely, and the architecture description including the HW/SW partitioning needs to be revised.* **DLO**

OLD: *Concrete plans of activities are thus still to be made, and methods and tools to be applied will have to be selected. Only the first phase of V&V activities is described in Sec. 8.3.1.* **DLO**

3 Verification & Validation in the Design Process

D2.3 defines the openETCS process on an abstract level. It already defines the main steps. A slightly more detailed picture than the one given in D2.3 is given in Fig. 1.

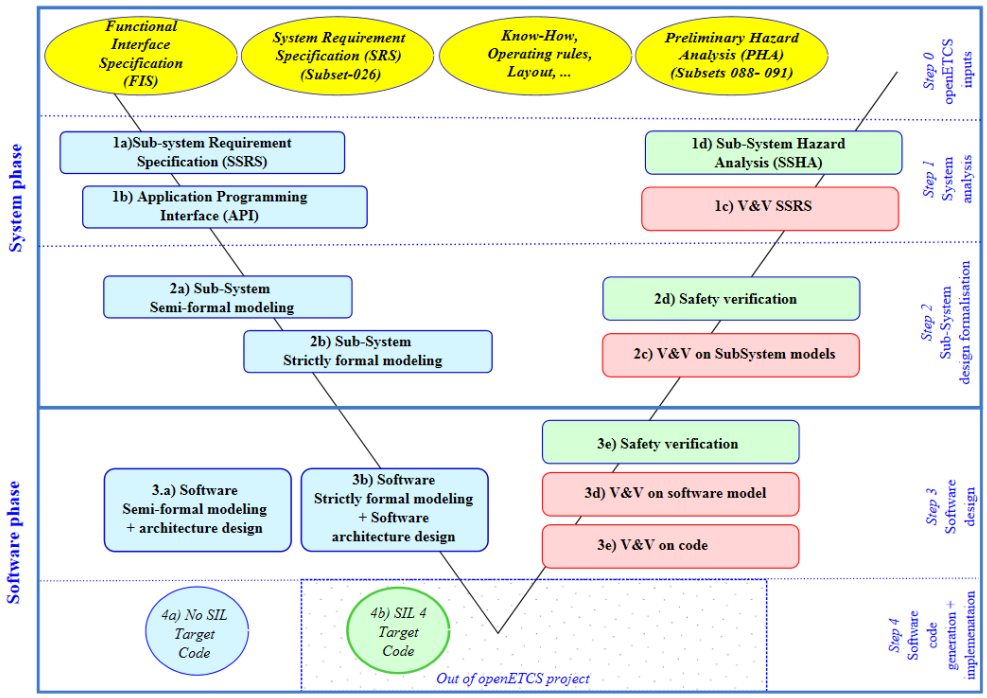


Figure 1. openETCS Process (rough view)

As a basis for planning the V&V activities, the process sketch permits to name the main phases. Planning will need a better definition of the stages (scope of the design artifacts, level of detail, respective system boundaries), detailed planning a specification of the artifacts to be produced. Version V01 of this document will in these respects be of an accordingly preliminary nature.

Part I

Verification & Validation for a Full Development

OLD: *W.r.t. verification & validation, openETCS shall achieve two goals:*

1. *Design a tool-supported method with which the EVC software can be developed and maintained so that it is suitable for integration in (SIL-4) certified products. This will be described in Sec. 4, "Verification & Validation Strategy for a Full Development".*
2. *Perform part of the development (including tool and method evaluation and the study of tool qualification questions) on representative parts of the design. The plans for that will be detailed in Sec. 7, "Verification & Validation Strategy for openETCS".*

DLO

OLD: *In more detail, the verification and validation have to consider the following aspects throughout the development process.*

- *functionalities of the system and the sub-system,*
- *system and sub-system architecture,*
- *external and internal interfaces of the sub-system,*
- *software components,*
- *performance and Safety objectives and constraints,*
- *functional properties,*
- *safety properties.*

DLO

4 Verification & Validation Strategy for a Full Development

The overall strategy is to support the design process as specified in D2.3 and its partial instantiations within openETCS. In accordance with the project approach, V&V shall be done in a FLOSS style, and it has to suit a model-based development. A further main consideration shall be to strive for conformance with the requirements of the standards (EN 50128 and further). Of particular importance in that respect is of course the interface to the safety considerations and the contribution of V&V to the safety case.

Here, the ideal shall be described: How will the V&V part of a FLOSS development of the open-source EVC software look like, what are its constituents, and how do they act together in developing and maintaining this software within the openETCS ecosystem.

4.1 Verification Strategy for a Full Development

This section defines the strategy for verifying a full development of the EVC software from the requirements source (ss 026+TSIs+...). This ends with the verification of the software/hardware integration. In current view, the API defines the interface and relevant properties of the hardware. Thus, SW/HW integration for openETCS will most probably done virtually with an instantiation of the API playing the role of the hardware.

4.1.1 System Test Campaigns

Testing is one of the main means for verification. The three main features of test campaigns which need to be specified are:

- Definition of the different campaigns for each design phase and their characteristics.
- Definition of the campaign objectives, and of the kind of test to be performed in order to achieve these objectives.
- Traceability matrix between characteristics to check/ validate and test cases.

4.1.2 V&V Actions Criteria

These criteria shall define the acceptance of system or component modification, and the interruption criteria shall define the condition under which a test campaign should be stopped for instance. These criteria have to bet set during the V&V plan redaction. During the verification phase, these items are verified thanks to the actions criteria list described in the subpart related to the development phases and items verification.

4.1.3 Organization Charts and Responsibilities

The plan shall define the roles and responsibilities of people involved in the V&V activities (according to standards and project needs). Must also be defined the teams compositions, and the

deliverables product owners, as well as the group and committee for modification. These items are verified during the verification phase.

4.1.4 Internal Means and Tools

The plan shall list and define the means and tools needed for V&V actions (also when they are needed). These items are verified during the verification phase.

4.1.5 External Means and Tools

The plan shall list and define the means and the tools used by the interfaces with the project, and which company or structure is using them. These items are verified during the verification phase.

4.1.6 Development Phases and Item Verification

Needs to be revised.

The plan for the development phases verification is based on requiement coverage at system and sub-system level. The requirements are usually decomposed and gathered in a table, and this table constitutes a road-map for verification activities, and is closely followed in order to figure out if the required items are covered, and if so, in which document. The following items give an overview of items checked during verification phases, regarding the kind of prescriptions checked, and the appropriate related development phase.

Introduction: This part gives an overview of Verification general purposes, of main verification phases (Safety Specification of Software, architecture, and overall test campaign), and verification support formalism (harmonized against different Verification activities, can be a simple table).

Software Security/Safety Prescriptions Specification Phase: Gives a list of prescriptions (specifications top level from a top level specification providing the customer needs), and trace their coverage (justification and in which document this justification is provided). Here is an example: “Prescription specification related to Software safety should come out from upper level system safety prescriptions and from prescriptions from Security schedule. This information should be communicated to the Software developer.”

Design Phases and Software Development: This phase is based on the same approach of the previous part, and prescriptions are gathered in a table. The different categories considered are defined in the following parts: (), , detailed design and development prescriptions, coding prescriptions, Software modular tests and Software integration tests prescriptions.

General Requiements: For instance: design representations shall be based on a clearly defined notations, or limited to clearly stated characteristics

Software Architecture Related Requirements: . For instance: Any System Safety prescription modification related to security should be documented and acknowledged by the developer.

Support Tools and Coding Languages Requirements: . For instance: According to the type of development, the compliance with the given prescriptions is under the software supplier responsibility.

Detailed Design and Development Prescriptions: For instance: “The Software shall be produced in order to insure the required modularity, testability and modification ease.”

Coding Prescriptions: For instance: “Each Software code module shall be reviewed”

Software Modular Tests Prescriptions: For instance: “Modular tests shall be documented”

Software Integration Tests Requirements: For instance: “Integration test specification shall encompass: integration groups easily manageable, test cases and test data, test kinds, environment, tools configuration on test program, test acceptance criteria, corrective actions procedures”

Software Security Validation Planning: The same formalism as previous parts can be used, and this part sums up the different requirements/specification for the planning regarding the safety activity. For instance: “Planning shall be managed in order to specify technical and process steps necessary to prove that the software is compliant with safety prescriptions”.

Programmed Electronic Components (Hardware and Software): This part is specifically related to the integration conditions. This part is not supposed to be considered in openETCS, as no integration is planned in the project so far.

Software Safety Validation: For instance: “Validation activities shall be performed as specified in the validation plan”.

Conclusion: (related to the requirements coverage). In our case, the prescriptions are the requirements explained in the WP4 (these prescriptions have to be refined from the different project inputs, such as CENELEC, SRS, FPP. . .).

4.2 Validation Strategy for a Full Development

Validation, according to the standard, starts after SW/HW integration. In this section, it shall be detailed how this should look like for the openETCS architecture approach (with SSRS and API). Ideal would be a description of how a full openETCS EVC software could be taken up by some manufacturer and brought to life in a product (validation aspect only, of course). Validation will use tests covering operational scenarios.

Not-so-classical validation can start earlier when executable models become available. If a model can be animated to run an operational scenario (perhaps with some additional environment/rest-of-system modeling), design defects may get unveiled before the real validation. This is, however, not an activity which is mentioned as a development activity in the standard EN 50128. Thus, to use results of “early validation” in a validation report requires a definition of its role and an argument for its usefulness.

4.2.1 Validation Case

The aim of the Validation Report is to demonstrate that the objectives of validation which have been set in the validation plan have been achieved. This concerns the adequacy of the software design documentation and that components and system behavior is compliant with the software requirements.

The inputs for validation at system level are:

- Software Requirement Specification (design, architecture),

- Sub-System Requirement Specification (encompassing the architecture, interface description and requirement allocation),
- Application Programming Interface,
- Internal and External constraints,
- Validation constraints list,
- Validation Plan,

The inputs for validation at software level are:

- Components Requirements Specification,
- Integration Specification,
- Integration Report,
- Test Specifications,
- Test reports.

The outputs are:

- Software Validation Report (Verdict on software ability to fulfill the objectives and functionalities defined in the requirement specification.),
- Software Validation Verification Report,

4.3 Safety Interface

There are two aspects: (1) Safety requirements come from a hazard and risk analysis. (2) The main contribution of verification & validation of the openETCS SW of the EVC for the safety case of the EVC (HW and SW) should be the validation report of the software. This has to prove that the software as implement achieves all its safety goals which have been assigned to it. These two aspects are adressed (partly) in D4.2.3. The following text should be adapted.

The safety activities in the software development process are closely connected to the verification & validation activities as these provide the overall system safety requirements, derive corresponding safety related design specifications and collect the verification & validation documentation to build the safety case. The safety activities are identifying unwanted accidents resulting in harm and analyses the potential hazards, which could lead to the harm. Resulting from this in depth analysis certain requirements are derived, which provide the overall safety goals for the system. If the initial risk of a hazard resulting in harm has to be reduced or the possibility of harm shall be eliminated at all specific safety designs are developed to reach the required risk. The verification & validation process has to verify these specifications and validated that the requirements hold for the developed software. Respectively, a complete documentation is needed to show in the safety case that this steps have been done according to the overall verification & validation plan.

Therefore three main interfaces to the safety activities have to be maintained in the verification & validation process:

- **Verification of safety design specification**: Based on risk control measures documented in the Hazard Log specific safety design specifications are derived and written in the backlogs for model and code development. As it is done for the overall model and code specification the verification activities have to demonstrate that all of the safety specifications have been implemented properly.
- **Validation of safety requirements**: The overall software validation has to demonstrated that the safety goals are met by the developed product. Therefore the safety requirements which have been stated based on potential accidents and accepted risk levels have to be validated. As these in many cases requires proof for the absence of certain conditions, it is important that verification & validation activities work in a close iterations with the safety activities to ensure that the safety requirements are stated in a way that can be validated.
- **Creation of verification & validation documentation**: As the safe case has to provide the complete argumentation that all needed steps to ensure a qualified and safe development process have been performed properly, it is important that the verification & validation plan and all resulting verification & validation reports are coherent and allow to show a close chain of arguments.

5 Verification Plan for a Full Development

The section is going to instantiate the generic Verification & Validation plan from the standard to the EVC development in openETCS-style (FLOSS). This entails the organisation, a definition of the requirements, generic schedule covering all design steps, resources, responsibilities, tools, techniques, and methodologies to be deployed in order to perform the verification activities, and all the documents which are to be produced.

The result must conform to the requirements of the standards for a SIL 4 development.

As D2.3 gives only a rough description of the development steps and not yet a complete list of design artifacts, nor one of methods applied and formats to be used, this first version (V01) of the V&V plan will also lack detail which will to be added in later revisions as these informations become more concrete.

5.1 Verification & Validation Plan Overview

This section gives an overview of the verification & validation plan for a full development.

5.1.1 Verification & Validation Organisation

This section defines the relationship of verification and validation to other efforts such as development, project management, quality assurance, and configuration management. It defines the lines of communication within the verification & validation, the authority for resolving issues, and the authority for approving verification & validation deliverables. Here, the verification & validation aspect of the “openETCS-ecosystem” should be outlined.

5.1.2 Verification & Validation Activity Overview

This section gives a short overview of the activities (Verification or Validation) which happen at the respective development steps, to be detailed in the subsequent sections. The numbering (e.g. 2e) refers to Fig. 1. Abbreviations used are defined in the glossary, Sec. A.3.

SSRS—Verification (1c): verification that the SSRS the requirements consistently extends the requirements base.

SSRS—Validation (1c): Deriving a sub-system test specification

SFM—Verification (2c): Verification that the model formalises the requirements

SFM—Validation (2c): Detailing the test specification, perhaps validating the model (e.g. via animation)

SW-SFM—Verification (3d): Verifying the SW-HW architecture definition (should be somewhere) and the software model

SW-SFM—Validation (3d): Perhaps validation of the software model

SW-FFM—Verification (3d): verification, employing also formal methods/tools

SW-FFM—Validation (3d): validation, may e.g. employ model checkers

Code—Verification (3e): verification depends on the code generation method (manual, generated, generated with validated tool), unit test requirements have to be met, afterwards code integration tests

Code—Validation (3e): no specific activities foreseen

The following step descriptions are preliminary and shall serve as a concept which is to be detailed in later revisions (V02 and up).

EVC Software—Verification (tbd): Perform software system verification

EVC Software—Validation (tbd): Validation against user requirements/scenarios, broken down to software functionality

SW/HW integration (tbd): Use the API in a simulation environment as a replacement of actual SW/HW integration

Final Validation (tbd): Apply user (railway operator) requirements and scenarios (based on the sub-system test specification)

5.1.3 Schedule

The overview of the activities given above shalle be detailed in this section. The objective here is to define an orderly flow of material between project activities and verification tasks.

5.1.4 Verification & Validation Resources

In a regular development project, resources needed to perform verification tasks, including staffing, facilities, tools, finances, and special procedural requirements such as security, access rights, and documentation control, have to be defined. Here, where we define merely a pattern of a V&V plan, the emphasis will lie on spelling out requirements and proposing principal solutions.

5.1.5 Responsibilities

Table 1. General Verification & Validation Responsibilities

Role	Name of the person	Affiliation	Activity Code
Verification Team Manager			
Verifier			

5.2 Requirements Base

This section shall provide references to all requirements against which the design is to be verified and validated. It does not include process requirements. For the latter, see Sec. A.

The requirements on the EVC software origin in the SS-026 and TSI specifications.

5.3 Verification Acitivities for a Full Development

for each of the verification steps identified in the plan overview, the following has to be instantiated:

5.3.1 DAS2V Verification

5.3.1.1 Task

5.3.1.2 Documents to Be Produced

5.3.1.3 Phase Specific Activities

5.3.1.4 Techniques and Measures

Here the verification plan begins

5.3.2 SSRS Verification (1c)

5.3.2.1 Task

The SSRS (sub-system requiement specification) outlines the subsystem which is going to be modeled within the project. The SSRS describes the architecture of the subsystem (functions and their I/O) and the requirements allocated to these functions. If necessary, the requirements are rewritten in order to address the I/O and to correspond to the allocation. It also provides the classification into vital and non vital requirements and data streams. The architecture part is described in a semi-formal language, and the requirements are described in natural language.

The SSRS is to be viewed as a supplement to the SS-026 and the TSIs and is not intended to replace them. The verification has to check that a complete and consistent set of functionalities have been identified and that the architecture is adequate.

5.3.2.2 Documents to Be Produced

SSRS verification report.

5.3.2.3 Phase Specific Activities

5.3.2.4 Techniques and Measures

Due to the informal nature of the SSRS, mainly manual techniques are to be applied.

¿Review?

5.3.3 SFM Verification (2c)

5.3.3.1 Task

5.3.3.2 Documents to Be Produced

5.3.3.3 Phase Specific Activities

5.3.3.4 Techniques and Measures

5.3.4 System Verification

SSRS Verification(1c)

The SSRS Verification phase refers to the task already defined in the Sec. 5.3.2 of this document that involves the full development. For further details about the activities involved, please go to the mentioned section.

5.3.4.1 Software Verification

This section describes software verification activities in greater detail than the description above. This presentation of the material shall be unified in due course.

The SW process is detailed in the D.2.3 OpenETCS process. Bearing in mind that the tasks described in this openETCS Verification plan are strongly linked to the Software process defined in that document, the following figure is included in order to have present the key points to be covered in the plan.

SW Requirements Verification

Task

In the SW Requirements Specification the SSRS shall be taken as starting point, redefining it to ensure the software constraints are considered. The SW Requirements Verification phase then shall verify that the proposed Software Requirements cover as much as possible of the SSRS and provide a correct implementation of the System Requirements in the Software context. On the other hand, the Verification done in this phase shall include the assessment of the SW Requirements modelling, the objective is to ensure the representation of the requirements is coherent with their specification as well as complete, explicit and implementable, as well as traceable to the semi-formal and/or formal models defined in the system phase.

Documents to Be Produced

- SW Requirements Verification Report

Activities

Some activities shall be performed to ensure the requirements are written in a readable and testable manner. The Verification of the Software Requirements shall assess whether the requirements met the following aspects:

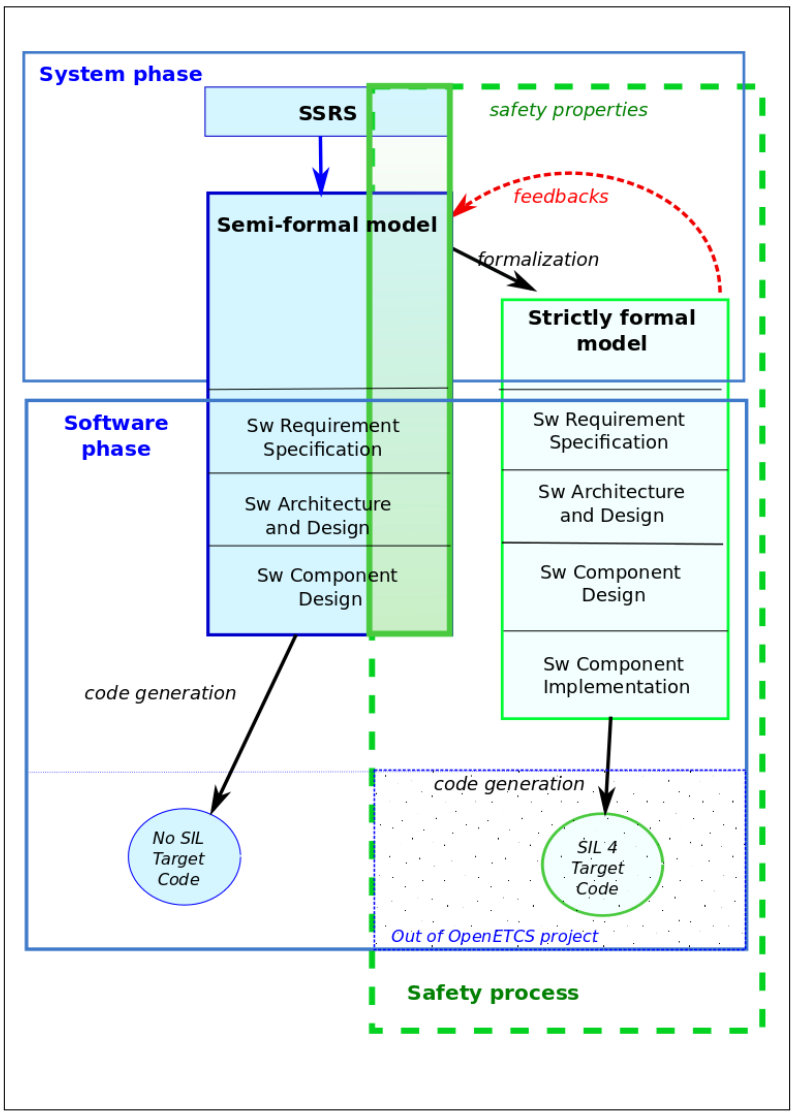


Figure 2. Software phase description

- Deterministic: Given an initial system state and a set of inputs, you must be able to predict exactly what the outputs will be.
- Unambiguous: All openETCS project members must get the same meaning from the requirements; otherwise they are ambiguous.
- Correct: The relationships between causes and effects are described correctly.
- Complete: All requirements are included. There are no omissions.
- Non-redundant: Just as the Software modelling (semi-formal and formal) provides a non-redundant set of data, the requirements should provide a non-redundant set of functions and events.
- Lends itself to change control: Requirements, like all other deliverables of the openETCS project, should be placed under change control.
- Traceable: SW Requirements must be traceable to each other, to the SSRS, to the objectives, to the design, to the test cases, and to the code.
- Readable by all project team members: The project stakeholders, including the users, experts and testers, must each arrive at the same understanding of the requirements.

- Written in a consistent style: Requirements should be written in a consistent style to make them easier to understand.
- Explicit: Requirements must never be implied.
- Logically consistent: There should be no logic errors in the relationships between causes and effects.
- Lends itself to reusability: Good requirements can be reused on future projects based on openETCS.
- Succinct: Requirements should be written in a brief manner, with as few words as possible.
- Annotated for criticality: Each SW requirement should note the level of criticality to the openETCS project. In this way, the priority of each requirement can be determined, and the proper amount of emphasis placed on developing and testing each requirement.
- Feasible: If the software design is not capable of delivering the requirements, then the requirements are not feasible.

The activities that shall ensure these aspects are successfully met are the following:

- *SWReq-Ver-Act1. Compliance with SSRS*: the SW Requirements are based on the SSRS, so it shall be ensured that their specification is compliant and coherent with regard to the SSRS; the SW requirements complement, adjust and enlarge the scope delimited by the SSRS in the Software phase.
- *SWReq-Ver-Act2. Accuracy, Consistency, Completeness, Correctness assurance*: A significant problem with recording requirements as text is the difficulty of analyzing them for completeness, consistency, and correctness. The Specification of the SW Requirements has precise syntactic and semantic rules (e.g., data type consistency) and the specification of the requirements shall be compared against those established rules to ensure these aspects are correctly covered.
- *SWReq-Ver-Act3. Testability assurance*: The two objectives for testing the openETCS safety-critical SW are the demonstration that the software satisfies its requirements and the demonstration that errors that could lead to unacceptable failures have been removed.
- *SWReq-Ver-Act4. Verificability assurance*: The SW Requirements shall identify the main functionality and describe the functional breakdown and data flows from top-level functions to low-level functions. The description provided as well as the flows identified shall be verifiable.
- *SWReq-Ver-Act5. Compliance with Standards*: Compliance with CENELEC Standards (EN50126, EN50128 and EN50129) shall be verified.
- *SWReq-Ver-Act6. Traceability with SSRS*: Each SSRS allocated to software must map to one or more software requirement. Traceability analysis, which can be automated, determines the completeness of the mapping of SSRS to software. A table similar to this one shall be obtained, either manually or automatically to assess the conformance with this expected activity.
- *SWReq-Ver-Act7. Requirements modelling correctness*: Considering "the result of the modelling activities is a document that represents a thorough understanding of the problem the

proposed software is intended to solve"[1. Cho, Chin-Kuei, Quality Programming, John Wiley & Sons, Inc, 1987, p 21.], during the SW Requirements verification, it shall be possible to capture all required information, including additional information for safety-critical aspects with the support of the models. These Requirements representations shall be compared to their original specification and assess whether they are complete and enough to cover all the information provided by them.

Table 2. Requirements Trace Table

Source SSRS ID	SW Requirement ID
SSRS-xxxx	SWR-yyy
SSRS-xxxx	SWR-yyy

Tools, Techniques, Methods and Measures

This section shall identify the special software tools, techniques, and methodologies to be employed by the verification team. The purpose of each should be defined and plans for the acquisition, training, support, and qualification of each shall be described in this section.

The following table summarizes the Techniques, methods, measures or tools proposed for the identified activities.

Table 3. SW Requirements Verification Tools, Techniques, Methods and Measures

Activity	Techniques/ Methods/ Measures	Tools
SWReq-Ver-Act1	This compliance is verified by peer review. The first version of SW Requirements is usually incomplete and composed primarily of top-level adjustments. The meaning of the SW requirements is described textually.	<To be defined>
SWReq-Ver-Act2	Since the modelling at this stage is provided, verification is mostly based on review and with a tools-based modelling approach. Some consistency checks between the SW Requirements and both the semi-formal modelling and the formal modelling can be automated.	<To be defined>
SWReq-Ver-Act3	Requirements-based testing has been found to be effective at revealing errors early in the testing process Requirement models created with standard UML use-case notations are not robust enough to create requirements based test cases that met the safety-critical aspects, so the standard use-case model notation shall be extended to allow the capture of additional information required to determine test cases.	Test cases can be automatically generated from "test-ready" use-case models. <To be defined>
SWReq-Ver-Act4	Verification is mostly based on review	<To be defined>

Continued on next page

Table 3 – continued from previous page

Activity	Techniques/ Methods/ Measures	Tools
SWReq-Ver-Act5	Verification is mostly based on review	<To be defined>
SWReq-Ver-Act6	A traceability matrix between SSRS and SW Requirements shall be prepared either manually or automatically and assessed by review	Integrated tools can generate software to system requirements traceability tables showing which software requirements are allocated to system requirements. <To be defined>
SWReq-Ver-Act7	Verification is mostly based on review	<To be defined>

SW Architecture, Design and Modelling Verification

Task

This phase aims to ensure that the software architecture design adequately fulfils the software requirements specification already verified in the SW Requirements Verification phase. The software architecture defines the major elements and subsystems of the openETCS software, how they are interconnected, and how the required (safety integrity) attributes will be achieved. It also defines the overall behaviour of the software, and how the software elements interact. To carry this phase the information from the current software lifecycle phase shall be verified. All essential information should be available and must be verified; the information should include the adequacy of the specifications, design and plans in the current phase. The architecture and design shall be modelled and the coherence of those models with regard to their specification shall be verified to assess whether the constraints are correctly treated. The verification configuration should be precisely defined and the verification activities shall be repeatable.

Documents to Be Produced

- SW Architecture, Design and Modelling Verification Report

Activities

Software architecture verification should consider whether the software architecture design adequately fulfils the SW requirements specification.

The essential properties to be verified during this phase are:

- What is the software supposed to do: including the SW specification and the system capabilities

- What is the software not supposed to do: be aware of the unexpected emergent behaviour, boundary specifications, inhibits
- What is the software supposed to do under adverse conditions: system biases, tolerances
- What are the safety considerations
- What are the integration or interfacing considerations: subsystems, modules, components, relationship between hardware and software, partitions, parallelism, compatibility
- What are the dependability considerations: performance, capacity, complexity, stability

Considering these aspects, a table shall be prepared to list a relation of verification dimensions regarding to the Architecture and Design that facilitates the process. An example is given below:

Table 4. SW Architecture Verification preparation table - Example of contents

Verification dimension	Intended behaviour	Unacceptable behaviour	Adverse conditions	Safety	Integration	Dependability
Performance analysis	Service delivery	Consequences of service failure, unexpected emergent behaviour	Distributed redundancy and fail-over			
Timing Requirements	normal behaviour	Blown performance margins, missing timing requirements	Distributed redundancy and fail-over	Safety and Recoverability, timing margins	Timeline management, phase transition interlocks	
Security Requirements	Secure communication	Compromised communication	Redundant communication channels			

The software architecture modelling, which consists of determining and connecting software components, requires a phase of analysis to be able to validate the representation carried out, as errors can occur in the representation. It is thus necessary to be able to identify and to provide them to the designer.

Modelling must also bring answers in term of feasibility and the constraint shall be analyzed so the interdependent functions should not overlap.

The verifications done at the software architecture level are related to specification and execution model coherency.

The expected activities for this phase are the following:

- *SWArch-Ver-Act1. Check the software architecture design*: The SW architecture and design should fulfil the SW requirements specification. From a safety point of view, the software architecture phase is where the basic safety strategy for the software is developed.
- *SWArch-Ver-Act2. Handle attributes*: The attributes of major elements and subsystems should be adequate with reference to the feasibility of the safety performance required, testability for further verification, readability by the development and verification team, and safe modification to permit further evolution. With the attributes, on which we rely once the software is specified, it is possible to check if the model of execution and the coherence of the periods of the various software components are corrects.
- *SWArch-Ver-Act3. Check incompatibilities*: The incompatibilities between design and specification should be checked.
- *SWArch-Ver-Act4. Model coherency*: Verify the representation carried out with the modelling. The errors identified can be related to the symmetry of the inter-connected functions. A certain number of constraints of coherence must thus be analyzed.
- *SWArch-Ver-Act5. Constraints analysis*: Ensure system dysfunctions do not occur. Some components can also execute an operation depending on a signal resulting from another component, which results in constraints of synchronization that must also be taken into account.

Tools, Techniques, Methods and Measures

The specific software tools, techniques, and methodologies to be employed by the verification team are to be identified here. The purpose of each should be defined and plans for the acquisition, training, support, and qualification of each shall be described in this section.

The following table summarizes the Techniques, methods, measures or tools proposed for the identified activities.

Table 5. SW Architecture, Design and Modelling Verification Tools, Techniques, Methods and Measures

Activity	Techniques/ Methods/ Measures	Tools
SWArch-Ver-Act1		<To be defined>
SWArch-Ver-Act2		<To be defined>
SWArch-Ver-Act3		<To be defined>
SWArch-Ver-Act4		<To be defined>
SWArch-Ver-Act5		<To be defined>

SW Component and Modelling Verification

Task

This phase analyzes whether the SW components defined in the openETCS Software process are verifiable by design, with a focus on the formal verification of data aspects of components and components modelling. This task requires a detailed understanding not only of the corresponding specification, but also the environments and methodology in which the verification component will be used.

This phase delivers methods and tools to

- Verify the component models by model-checking techniques, with a focus on behavioural aspects such as attributes and data-dependent properties
- Verify the specification of the components is coherent to the SW architecture and design previously verified. This activity shall be based on the existing models in the component stage as well as previous phases, in close connection with the model checking activities.

Documents to Be Produced

- SW Component and Modelling Verification Report

Activities

This phase aims to assess whether the specification and modelling of components are reusable, configurable, and implementable in the expected code automatically generated environment.

- Within the context of the models that contain the components identified for the SW phase, the verification of components within a control system model shall be required. As standalone components — For a high level of confidence in the component algorithm, verify the component in isolation from the rest of the components and sub-systems identified. This approach is called component analysis.
- Verifying standalone components provides several advantages:
  - The analysis can be be used to focus on portions of the design that cannot be tested because of the physical limitations of the system being controlled.
  - This approach can be used for open-loop simulations to test the plant model without feedback control.

The components shall be identified at the points of minimal coupling (minimal control and/or information exchange) and decompose the specification. The smaller components might be amenable to be verified and how to abstract these parts of the specification shall be analysed.

The rest of the specification could be exposed to formal analysis, such as model checking or theorem proving. The abstraction reduces the number of states that need to be checked by an automated verification technique and helps in avoiding the state explosion problem, which occurs in traditional model checking.

An approach to specification decomposition is slicing the system specifications represented with Petri Nets. In this way, the specification shall improve the understanding of the complexity of the system and high-risk components can be identified earlier.

The activities that shall ensure these aspects are successfully met are the following:

- *SWComp-Ver-Act1. Verify components with minimal coupling independently of the model*: Within this activity a component analysis shall be performed to verify model blocks, atomic subsystems and statechart atomic subcharts. After extracting the blocks, or contents of the subsystems and subcharts, a harness model shall be created for the referenced model and the extracted model with the contents of the subsystem o subchart. A Test suite shall be prepared and executed to record coverage and output values. The Code Generation Verification (CGV) shall be invoked to execute the test cases on the generated code for the model that contains the component.
- *SWComp-Ver-Act2. Verify components with minimal coupling in the context of the model*: A similar process shall be conducted in this case but performing a system analysis to verify the model blocks in the context of the model.
- *SWComp-Ver-Act3. Formal analysis of components*: Perform model checking of components against properties; the results obtained shall highlight that: the component satisfies the properties for the environment; the component violates the properties for the environment; characterization of those environments in which the components satisfied each property.

Regarding the Modeling Verification, the activities involved shall ensure that:

- The models have been designed correctly and covers all the parts involved in the openETCS project
- The algorithms have been implemented properly
- The models do not contain errors, oversights, or bugs
- The specification is complete and that mistakes have not been made in implementing the model

Each of the models defined in the full development context shall be developed for a specific purpose and its validity is determined with respect to that purpose. It shall be considered that a model that may be valid for one set of conditions, can be invalid in another; because of that, the model's output variables of interest shall be clearly identified and their required amount of accuracy be specified.

It is costly and time consuming to determine whether a model is valid over the complete domain of the full development context. Instead, tests and evaluations are conducted until sufficient confidence and certainty is obtained that a model can be considered valid. In the model verification activities, tests are performed, errors are identified, and corrections are made to the underlying

models while the models are exercised for the identified cases. Furthermore, the verification of the models can result in retesting requirements to ensure code integrity.

If the verification process determines that a model does not have sufficient accuracy for any one of the sets of identified conditions, then the model is invalid. However, determining that a model has sufficient accuracy for numerous conditions does not guarantee that a model is valid everywhere in its applicable domain.

- *SWModel-Ver-Act1.*
- *SWModel-Ver-Act2.*
- *SWModel-Ver-Act3.*

Tools, Techniques, Methods and Measures

The specific software tools, techniques, and methodologies to be employed by the verification team are to be identified here. The purpose of each should be defined and plans for the acquisition, training, support, and qualification of each shall be described in this section.

The following table summarizes the Techniques, methods, measures or tools proposed for the identified activities.

Table 6. SW Component and Modelling Verification Tools, Techniques, Methods and Measures

Activity	Techniques/ Methods/ Measures	Tools
SWComp-Ver-Act1		<To be defined>
SWComp-Ver-Act2		<To be defined>
SWComp-Ver-Act3		<To be defined>
SWModel-Ver-Act1		<To be defined>
SWModel-Ver-Act2		<To be defined>
SWModel-Ver-Act3		<To be defined>

SW code generation Verification

Task

This phase analyzes whether the generated code from the model is correct, robust, complete, coherent and reliable, with a focus on the formal verification and on the equivalence of the model and the generated code.

This phase delivers methods and tools to

- Verify the equivalence between the model and the generated code by equivalence testing and dynamic testing techniques
- Verify the correctness of the translation algorithm and its implementation
- Verify the complexity, coding guidelines and syntax of the generated code by static analysis techniques
- Verify the robustness of the generated code

Documents to Be Produced

- SW Code Generation Verification Report

Activities

When in the other phase the confidence in the correctness of the model has been verified, it is necessary to have high confidence in the generated code.

To achieve verified code generation, it is not sufficient to only verify the code generation algorithm due to the implementation of the algorithm might introduce errors as well. It is necessary to distinguish between the correctness of the translation algorithm itself and the correctness of its implementation.

Within the context of the Code Generation Verification, the code generation verification approach shall utilize translation testing to demonstrate that the execution semantics of the model are preserved during production code generation, compilation and linking

The validity of the translation process, i.e. whether or not the semantics of the model have been preserved during code generation, compilation and linking, shall be determined by comparing the system reactions

The activities that shall ensure these aspects are successfully met are the following:

- *SWCode-Ver-Act1. Semantic Properties Verification*: For safety reasons, it is necessary that the generated code is semantically equivalent to the original model specification to ensure correct software and system behavior.

  During this activity, the semantics of the original model is preserved during the code generation shall be demostrated. This will verify that the transformation algorithm is correct.

  To prove the semantical correctness of such a transformation, a suitable model semantics is needed.

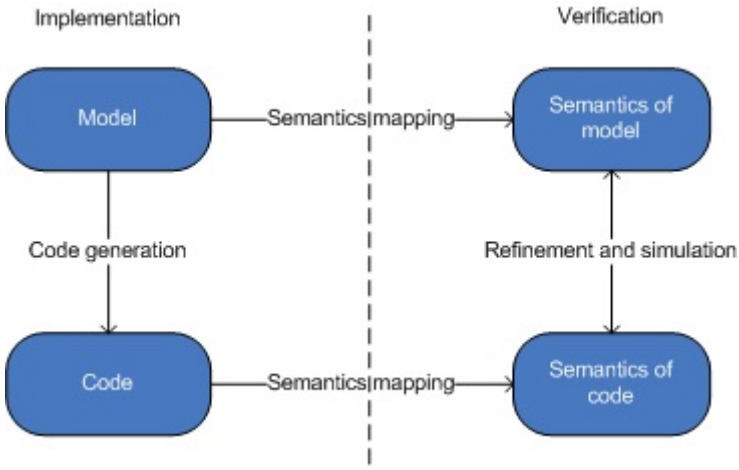


Figure 3. Transformation Verification

As a basic prerequisite, the semantics of the model and code must be comparable and it is also required a semantics for both model and code with the same semantic domain.

The first steps will be define the commom both model and code semantics. A multitude of approaches can be taken to define such a semantics. This definintion of the semantics poses subtle difficulties due to inherent ambiguities. Once this is done the coverage over semantics shall be identified.

Then, it will be necessary to establish a correspondence between the model and code, based on refinement, and to prove it by simulation.

A refinement mapping correlating the control points of the model and code, and indicating how the relevant model variables correspond to the code variables or expressions at each control point shall be established. For some transformation optimizations it is often impossible to apply the refinement based rule since there are often no control points where the states of the model and code can be compared. So, a large class of these optimizations shall be identified to allow their effective transformation verification.

After this, the simulation method shall be used for transformation verification.

This transformation from model to code will be considered semantically correct if the semantics of model and code are always the same.

Using the simulation principle to show that are equal, a simulation relation in which they are contained shall be found and defined. Note that the definition of a simulation relation is an artificial construct for conducting proofs.

- *SWCode-Ver-Act2. Functional Correctness Verification*: Within this activity, as a first approach for functional properties correctness verification, the automatic test cases generated from the model will be executed under the automatic generated code from the same model, so that, the coherence, functionality and coverage of the code shall be verified.

  Taking into account the first obtained results, it will be possible to check the coverage of both code and test cases, detect the necessary functional properties not generated automatically, and they shall be inserted manually into the code.

  When discrepances are found, an analysis of both test cases and code shall be done to check in which part the problem is taking into account the techniques and tools of the specific verification phase activities.

- *SWCode-Ver-Act3. Quality Properties (Complexity, Syntax, coding guidelines) Verification*: Within this activity, a static analysis shall be performed locating potentially vulnerable code.

The static analysis involves no dynamic execution and will detect possible faults such as unreachable code, undeclared variables, parameter type mismatches, uncalled functions and procedures, possible array bound violations, standards fulfillment i.e. MISRA standard, etc.

The static analysis techniques will also include:

- data flow analysis: it consists in a technique to follow the path that various specific items of data take through the code, looking for possible anomalies in the way the code handles the data items
- control flow analysis: technique for determining the control flow of the code. This type of analysis will identify infinite loops, unreachable code, cyclomatic complexity that identifies the number of decision points in a control flow, other complexity metrics like number of nested statements, etc.

With this analysis, errors in the structure and syntax of the code are going to be found but run-time errors (errors that only occur when the code is executed) will not be detected as we are not running the code.

Therefore this will not find things like memory leaks or check whether a value in a variable is correct.

- *SWCode-Ver-Act4. Equivalence Verification*: This activity shall use equivalence testing and other techniques like dynamic testing to demonstrate equivalence between the model and the generated code.

  During this phase, the execution of the model used for code generation and the object code derived from it with the same input stimuli followed by a comparison of the outputs shall be performed. The validity of the translation process, i.e. whether or not the semantics of the model have been preserved during code generation, compilation and linking, is determined by comparing the system reactions (results, which are the outputs resulting from stimulation with identical timed test inputs) of the model and the generated code.

  If the coverage achieved with the existing test inputs is not sufficient w.r.t. the selected model coverage metrics, additional test inputs that execute the model elements not covered shall be created. In practice, the tester can iteratively extend the set of test inputs using model coverage analysis until the chosen level of model coverage has been achieved. If full coverage for the selected metric(s) cannot be achieved, the uncovered parts shall be assessed and justification for uncovered parts shall be provided.

- *SWCode-Ver-Act5. Robustness Verification*: Within this activity a code analysis shall be performed to verify its robustness so it behaves reasonably in exceptional situations; wrong execution paths, computational errors or erroneous inputs and memory alterations coming from the environment(errors caused by interactions between the software and its environment(hardware and software))

  The exceptional situations coming from the software itself can be automatically proven using static analyzers based on abstract interpretation such as Frama-C or RSM

  After this analysis, there will be necessary to check the exceptional situations coming from the environment. The inconsistencies between the specification and the user constrains present in the code shall be detected. To ensure the robustness of a piece of code, the value of each un-trusted input must be checked against its correctness domain after its production and before its consumption

  The robustness review of the generated code can be done by manual code review to verify that all checks are present at right locations and that the checked domains are strictly included in the correctness domains.

Tools, Techniques, Methods and Measures

The specific software tools, techniques, and methodologies to be employed by the verification team are to be identified here. The purpose of each should be defined and plans for the acquisition, training, support, and qualification of each shall be described in this section.

The following table summarizes the Techniques, methods, measures or tools proposed for the identified activities.

Table 7. SW Code Generation Verification Tools, Techniques, Methods and Measures

Activity	Techniques/ Methods/ Measures	Tools
SWCode-Ver-Act1		<To be defined>
SWCode-Ver-Act2		<To be defined>
SWCode-Ver-Act3		<To be defined>
SWCode-Ver-Act4		<To be defined>
SWCode-Ver-Act5		<To be defined>

Traceability matrix Verification

Task

The Traceability Matrix is a resource to ensure that the project's scope, requirements, modeling and generated code remain as are expected to be when compared to the objectives to be reached. The matrix traces the different elements by establishing a thread for each requirement to the corresponding model and component, the generated code that consists on its implementations and the test cases defined.

- The specifications shall be verified following the traces between the elements defined in the matrix
- With the matrix verification it shall be ensured that all the required information is included in the specification, such as process models and data models
- Requirements that are not addressed by configuration items during design and code reviews can be identified; in a similar way, extra configuration items that are not required can be highlighted
- The verification of the matrix shall provide input to change requests and future project plans when missing elements are identified

Documents to Be Produced

- Traceability Matrix Verification Report

Activities

Taking the time to cross-reference each element involved in the project to another element ensures that the results obtained by each phase of the project are consistent with the requirements, the modeling and the code.

To accomplish this objective it shall be verified that the traceability matrix is in the following format:

- The matrix generated is a two-dimensional table,
- There is one column per identified element (such as requirement, model, components, or code),
- A check mark at the end of the table shall be put if all the row is well traced, coherent and unambiguous
- Various traceability matrices may be utilized throughout the project life cycle.
- At least the matrices shall contain the following elements:
  - Formal specification of Requirements: It shows that each requirement (obtained from SSRS) has been covered in an appropriate section of the formal specification.
  - Design specification through modeling to functional specification, this verifies that each function has been covered in the design.
  - System test plan to functional specification ensures the test case or test scenario for each process, component and each requirement in the functional specification have been identified

Tools, Techniques, Methods and Measures

The specific software tools, techniques, and methodologies to be employed by the verification team are to be identified here. The purpose of each should be defined and plans for the acquisition, training, support, and qualification of each shall be described in this section.

The following table summarizes the Techniques, methods, measures or tools proposed for the identified activities.

Table 8. Traceability Matrix Verification Tools, Techniques, Methods and Measures

Activity	Techniques/ Methods/ Measures	Tools
SWMatrix-Ver-Act1		<To be defined>
SWMatrix-Ver-Act2		<To be defined>
SWMatrix-Ver-Act3		<To be defined>

Test Verification

Task

Verification of the test suites designed for testing, the conformance of the implementation of the specifications, the design, the models and/or the code generated against the formal description of the openETCS system shall involve the verification of some relevant aspects of the test cases such as:

- expected input/output of the test behaviour
- test results
- test purpose

The properties of the test cases shall be defined as they were liveness properties, providing notations to specify the test purpose formally. All these properties expressed with formal notation can be verified using model checking, for example, on an extended state machine diagram, so the behaviour of the test case is represented. With this methodology, errors in the specification of the test cases can be found easily.

Documents to Be Produced

- Test Verification Report

Activities

There are two approaches when designing a test suite:

- *Semi-automatic generation of test-cases*: the formal specification of the system is used to generate the test cases. Previously there has been identified some test design techniques that generate test cases. However there could be some features, like the robustness capabilities of implementation, that cannot be derived to test cases using semi-automatic techniques.
- *Human design of test cases*: human designed test suites have some advantages over the semi-automatic ones that shall be taken into account, for example that the test case can be designed with a specific test purpose, the test cases can be grouped into various categories (basic interconnection tests, capability tests, valid behaviour tests, robustness/invalid behaviour tests, etc.) and the test cases can be manually designed for complex architectures. On the other hand these test cases, due to they have been designed manually, can be error-prone.

The test suites designed during the openETCS project shall combine both strategies, so in general terms some of them shall be generated semi-automatically following different techniques to be reviewed and adjusted later by the test engineers to ensure their appropriateness, and for the most specific contexts, the required experience and knowledge shall involve a manual design and a methodology to verify the correctness of those test cases against the formal specification.

Considering these aspects when designing the test suites, the following activities shall be done to verify the correctness and completeness of the tests cases generated:

- *SWTest-Ver-Act1. Verification of semi-automatic generated test cases*: The verification of the test case shall imply to check whether the essential information has been correctly generated. For example: are sufficient input combinations exercised?, are the tests complete and enough to find defects?, the conditions have been collected in the test cases, the test cases test one specific thing each to simplify the tracking of errors and obtain high coverage, the test cases say how the system works, the test cases are correctly organised in test suites and they keep the consistency so it will be easy to locate and add new test cases in the future, and finally assess whether the test cases are:
  - Fast: The test cases shall be fast to execute, the running should not take much time.
  - Independent: The test cases can be run in any order.
  - Repeatable: The result of the test case should be always the same, no matter how many times it has been executed.
  - Small: Small test cases are easy to understand and change, are also likely to be faster.
  - Transparent: It should be clear what the purpose of each test case is.
- *SWTest-Ver-Act2. Verification of manual designed test cases*: the verification process is similar to the process already defined in the verification of semi-automatic generated test cases activity. Besides this, some aspects related to the manual specification of test cases shall be taken into account like the peer-review done to find typical human errors, robustness and completeness checking or general coherence.

Tools, Techniques, Methods and Measures

The specific software tools, techniques, and methodologies to be employed by the verification team are to be identified here. The purpose of each should be defined and plans for the acquisition, training, support, and qualification of each shall be described in this section.

The following table summarizes the Techniques, methods, measures or tools proposed for the identified activities.

Table 9. Test Verification Tools, Techniques, Methods and Measures

Activity	Techniques/ Methods/ Measures	Tools
SWTest-Ver-Act1		<To be defined>
SWTest-Ver-Act2		<To be defined>
SWTest-Ver-Act3		<To be defined>

Coverage Verification

(futher development: requirements, model and code coverage description in task and activities)

Task

Test coverage is a measure of test effectiveness and it aims to reach a sufficient fault removal with contained testing effort. Considering higher levels of coverage are associated with better quality, this aspect raises an important practical question of how hard it may be to increase coverage.

The test coverage shall be influenced by:

- Modeling complexity
- Code Complexity
- Type of functionality
- Testing team experience in the area to be covered

All these factors were related to the level of coverage and quality, with coverage having an effect even after these adjustments. Furthermore, the test effort increases exponentially with test coverage, but the reduction in field problems increases linearly with test coverage. This suggests that for most cases the optimal levels of coverage are likely to be well short of 100%. The level of coverage needed shall be analysed in each phase of the project and adjusted when more information and details are obtained.

Documents to Be Produced

- Coverage Verification Report

Activities

One of the key tasks to take into account in the preparation of the V&V plan is that the correctness of openETCS system shall be verified with respect to its specification, while is is also assessed how complete the specification is and whether it really covers all the expected behaviours. Doing an exhaustive verification process is a great challenge considering we are speaking in terms of simulation-based verification, where test suites shall be prepared with finite subsets of input sequences.

In this context, it is essential to measure the exhaustiveness of the test suite with the support of numerous coverage metrics that reflect different aspects of the appropriateness of the test suite designed. The activities to be done to measure the coverage shall be focused not only on state-based coverage, but simulation-based verification to the formal verification setting as well.

Then, coverage metrics are used in order to monitor progress of the verification process, estimate whether more input sequences are needed, and direct simulation towards unexplored areas of the design and modeling. The metrics proposed shall measure the part of the design that has been activated by the input sequences.

The basic approach to coverage in testing, which is recording which parts of the design were exercised during the execution, cannot be used in formal verification because formal methods are exhaustive, and this shall be taken into account when designing the coverage strategy and dealing with formal methods.

The activities that shall ensure these aspects are successfully met are the following:

- *SWCover-Ver-Act1. Coverage in model checking*: Important advantages of model based testing are formal test specifications that are close to requirements, traceability of these requirements to test cases, and the automation of test case design. In model checking, coverage is a standard measure in testing, but is considerable difficult to compute in the context of formal verification. With this activity the correctness of the system shall be verified with respect to the desired behaviour by checking whether the involved models satisfy the formal specifications. The correctness and exhaustiveness of the specifications made shall have direct influence in the exhaustiveness of the model checking; the coverage metrics are the way to check this exhaustiveness and address the verification process to unexplored areas of the design. One technique to be applied in the activity shall be to measure coverage checking whether each output is fully determined by the specification, given a combination of input values. There are other possibilities, like do a model check for each of the mutant designs, and although a way to simplify the process is to use non-deterministic variables for mutations, this technique is still time consuming and costly, so it can be proposed for the most critical scenarios. The following sub-activities shall be included.
  - *SWCover-Ver-Act1.1. Coverage of the properties defined:* Model Checking is a method for deciding whether a given design satisfies a given formal property: in case the design violates the property, the Model Checker provides a trace that demonstrates how the property can be violated. On the other hand, when the answer to the Model Checking query is positive, most tools terminate with no further feedback. However, there can be situations where the properties are not as complete as expected and here the skills of the verification engineer are essential. For this reason, a sanity check for the completeness of the set of properties is to measure the coverage of them.
  - *SWCover-Ver-Act1.2. Code-based coverage metrics*: This activity shall imply the execution of syntactic coverage that shall be supported with the syntactic representation of the design with respect to which the coverage is measured. It covers the measures identified to assess the degree to which the source code obtained by automatic generation from the models is tested by particular test suites.
    Among others, the number of code lines executed during the simulation shall be measured (statement coverage), as well as the functions executed (function coverage), whether all the branches defined in each control structure present in the code (branch coverage) are executed at least once, the points of entry and exit are invoked at least once (decision coverage), the boolean sub-expressions (condition coverage), or combination of some of them like condition/decision coverage where both decision and condition coverage are checked to assess whether both have been satisfied in the generated code.
  - *SWCover-Ver-Act1.3. State-based model coverage*: It measures all the states that have been reached and checked. It is the essential measure to take in formal verification. The approach proposed is to combine coverage criteria, to use model transformations for testing, and to combine state machines with the other test models in order to reach a whole overview of the correctness and effectiveness of the models and whether these specifications are close to requirements.
  - *SWCover-Ver-Act1.4. Semantic coverage*: Semantic coverage metrics measure the part of the functionality of the design exercised by the set of input sequences. In this

context, the effect of the mutations shall be checked mainly on the satisfaction of the specification. The influence of mutations and omissions shall be checked on the result of model checking of the specifications, in such a way that the influence of omission or the changes of values of output variables shall be assessed on the satisfaction of the specification in the design.
Besides this, in path coverage the influence of omitting or mutating a finite path on the satisfaction of the specification in the design shall be assessed. Within this task, all possible mutations can be introduced consistently on each occurrence of the mutated element, on exactly one occurrence, or on a subset of occurrences, thus resulting in structure, node, or tree coverage, respectively. The application of mutations shall be done, as said before, in the most critical elements of the design.

- *SWCover-Ver-Act2. Functional Test Coverage*: It refers to metrics that create reports on the measurement of functionality exercised in the design and models obtained. In this area the metrics proposed are the control-flow coverage that indicates how completely the logical flow of the functional model is traversed, and the data path coverage, that indicates how completely the data paths have been exercised over a specified set of observable values. A list of assertions referring to the variables of the design shall be obtained, describing those assertions the conditions that may be satisfied during the execution or a state of the design. These kind of coverage metrics shall measure what assertions are covered by a given set of input sequences.

- *SWCover-Ver-Act3. Simulation-based verification*: Each of the metrics identified for the Simulation-based verification is addressed to a specific representation of the design or a specific verification goal. This activity shall be conducted taking into consideration the syntactic coverage and semantic coverage metrics perspective and shall contain the following sub-activities:

  – *SWCover-Ver-Act3.1. Syntactic coverage metrics*: Syntactic coverage metrics shall assume a specific formalism for the description of the design and this kind of metrics shall also measure the syntactic part of the design when executing a given input sequence. In this context, it shall be considered as a precondition to reach a high degree of coverage according to syntactic-based metrics before moving to other type of coverage metrics to apply when performing Simulation-based verification.
  The syntactic coverage metrics, as in the Model Checking perspective, shall include the Code coverage; the task shall be conducted in a similar way with statement and branch coverage as the basis, where the coverage is calculated on the basis of when these items (branches or statements) are executed at least once during the execution of a sequence. The expression coverage shall be also applied to check boolean expressions.

  – *SWCover-Ver-Act3.1. Semantic coverage metrics*: Semantic coverage metrics require the support of the experts involved in the design of the system and are more sophisticated than syntactic coverage metrics. Similarly to code coverage, a state or a transition shall be covered if it is visited during the execution of the input sequence. Limited-path coverage metrics shall be applied to check what expected sequences of behaviour are exercised; on the other hand, transition coverage shall be done as a special case of path coverage, but for paths of length 1. As said before, mutation coverage is the metric that inspired some important work on coverage in model checking; and it can be applied in the contexts of semantic coverage and in this specific case, for simulation-based verification. In mutation coverage, a small change or mutation ins introduced to the design, and it shall be checked whether the change leads to an erroneous behaviour. The coverage shall be measured in terms of the percentage of the mutant designs that fail. The goal here shall be to find a set of input sequences such that for each mutant design there exists at least one test that fails on it.

Low coverage indicates a possible incompleteness in the specification, which may lead to missed bugs in the non-covered parts of the design, so the levels of coverage shall be continuously monitored in order to assure the correct flow of the project.

Tools, Techniques, Methods and Measures

The specific software tools, techniques, and methodologies to be employed by the verification team are to be identified here. The purpose of each should be defined and plans for the acquisition, training, support, and qualification of each shall be described in this section.

The following table summarizes the Techniques, methods, measures or tools proposed for the identified activities.

Table 10. SW Coverage Verification Tools, Techniques, Methods and Measures

Activity	Techniques/ Methods/ Measures	Tools
SWCover-Ver-Act1		<To be defined>
SWCover-Ver-Act2		<To be defined>
SWCover-Ver-Act3		<To be defined>
SWCover-Ver-Act4		<To be defined>
SWCover-Ver-Act5		<To be defined>
SWCover-Ver-Act6		<To be defined>

Safety Verification

5.3.4.2 Tool chain Verification

Verification of the identified list of tools that have been collected in the tool chain.

5.4 Verification Reporting

This section describes how the verification activities have to be documented. Verification reporting will occur throughout the software life cycle. The content, format, and timing of all verification reports shall be specified in this section.

The following kinds of reports will be generated during the verification process:

- **Anomaly reports:**
- **Phase Summary Verification reports:**
- **Final report:**

5.4.1 Structure of the Verification Report

The following sketches the structure of the verification report. It shall cover the following central topics:

Header containing all information to identify, this report, the authors, the approbation and reviewing entities.

Executive Summary giving an overview of the major elements from all sections.

Problem Statement describing the challenges to be answered by Verification & Validation as well as the decisions to be taken based on the V&V results as well as how to cope with potentially faulty output. It further describes the accreditation scope based on the risk assessment done on V&V-level.

V&V Requirements Traceability Matrix links every V&V artifact back to the requirements to measure e.g. test coverage and to directly link V&V results to the requirements.

Acceptability Criteria, describing the criteria for acceptance of the artifact into the Verification & Validation process e.g. as the direct translation of the requirements into metrics to measure success, are used e.g. for burndown charts within the process.

Assumptions that are identified during the design of the verification and validation strategy and how these assumptions have an impact on the verdict by listing capabilities and limitations.

Risks and Impacts that come across the execution of V&V tasks together with the impacts foreseen.

V&V Design states how the V&V process builds up including data preparation, execution and evaluation.

V&V Methodologies giving a step-by-step walkthrough of all possible V&V activities including the assumptions, and verdict-relevant limitations and criteria for, e.g., model verification, model-to-code verification, unit testing, integration testing and final validation (according to the standard, this involves running the software on the target hardware).

V&V Issues describing unsolved V&V issues and their impact on the affected proof or verdict.

Peer Reviews going into details on how the community can take part and how official bodies and partners are integrated into the development and review process.

Test Plan Definition going into the details of testing by describing among other things:

- **Title** as a unique identifier to the test plan.
- **Description** of the test and the test-item giving information about version and revision.
- **Features** to be tested and not to be tested in combination are listed together with information background.
- **Entry Criteria** which have to be met by the EVC before a test can be started, e.g. that the EVC has to be in level 3 limited supervision with the order to switch to level 2.
- **Suspension criteria and resumption requirements** are the central key to a smooth automation of the tests covering topics like *when exiting this test before step 10, which entry criteria does it comply to or which resumption sequence has to be executed to continue testing*.
- **Walkthrough** covering a step-by-step approach of the test plan.
- **Environmental requirements** going into the details of what is needed concerning the test environment, e.g. tools, adapter, data preparation.

Discrepancy Reports identifying the defects.

Key Participants describing the assignment and task for each role involved.

Accreditation of Participants describing who was accredited to which role during the Verification & Validation phase.

V&V Participants listing the partners participating in V&V activities,

Other participants including other interest groups such as reviewer by affiliate partners[1].

Timeline giving the timeline for the baselines as input to the V&V process and identifying when each artifact should be created.

An example of a summary table of verification activities performed is given in Table 11. Such a table should enter the verification report.

Table 11. SW Verification Activities Breakdown

Activity Code	Activity	Responsibility	Status/Link
Phase 0: SW Requirements Verification			
SWReq-Ver-Act1	Compliance with SSRS verification		
SWReq-Ver-Act2	Accuracy, Consistency, Completeness, Correctness assurance		
SWReq-Ver-Act3	Testability assurance		
SWReq-Ver-Act4	Verificability assurance		
SWReq-Ver-Act5	Compliance with Standards		
SWReq-Ver-Act6	Traceability with SSRS verification		
SWReq-Ver-Act7	Requirements modelling correctness		
Phase 1: SW Architecture and Design Verification			
SWArch-Ver-Act1	software architecture design verification		
SWArch-Ver-Act2	Handle attributes verification		
SWArch-Ver-Act3	incompatibilities checking		
SWArch-Ver-Act4	Model coherency verification		
SWArch-Ver-Act5	Constraints analysis verification		
SWArch-Ver-Act6	Traceability verification		

Continued on next page

[1] affiliate partners are non-funded companies who signed the project cooperation agreement and with it get read access to the repositories starting from incubation phase to contribute e.g. by reviewing

Table 11 – continued from previous page

Activity Code	Activity	Estimated Delivery	Real Delivery
SWArch-Ver-Act7	Compliance with Standards verification		
Phase 2: SW Component Design			

Also a summary of the tool verification activities is to be included, as indicated in Table 12.

Table 12. Tool Chain Verification Summary

Activity Code	Activity	Estimated Delivery	Real Delivery
Phase 0: Tool chain Requirements			
Phase 1:			

5.5 Administrative Procedures

This section identifies the existing administrative procedures that are to be implemented as part of the Verification Plan. Verification efforts consist of both management and technical tasks. Furthermore, it is the task of the SQA team to monitor whether the procedures as defined in the management plans ([1], [SCMP], [Review and Revision processes]) are followed.

5.5.1 Problem Report

The problem reporting procedure is described within the document Change/Problem Management Process.

Any problem, failure and error encountered during the review activities (QA. Verification, Validation, Assessment) planned in the software development life-cycle, problems reported by users and customers as well as change requests initiated by any of the system stakeholders will be reported and managed following the Change/Problem Management Process detailed in [governance] and through the Change/Problem Management Tool.

5.5.1.1 Task Iteration Process

Any change in the requirements (system, sub-systems, sw or components) require repeated verification and validation activities.

Once the change is accepted following the change/problem management procedure, the phases and items affected by it must be evaluated. These tests will be redesigned to reflect the change in the requirement and will be executed again.

In turn, a new analysis of the Software Integrity Level will involve the analysis of the activities requirements and documentation presented by the EN50128 standard and include such activities in the SVVP if necessary.

5.5.1.2 Deviation Process

The Quality Manager will be informed in the case of detection of a deviation regarding Verification Plan. In addition, he/she also be informed if it is deemed necessary by an amendment to the Plan, whether or not motivated by a deviation

The Quality Manager will report such incidents to the Project Managers and with whom shall act appropriately. All persons listed in the responsibilities section (Sec. 5.1.5) shall be informed of a change in the Verification Plan

5.5.1.3 Control Procedure

Control procedures are specified in the Configuration Management Plan [SCMP]

6 Validation Plan for a Full Development

For each of the validation steps identified in the plan overview, the following has to be instantiated:

6.1 DAS2V Validation

6.1.1 Task

6.1.2 Documents to Be Produced

6.1.3 Phase Specific Activities

6.1.4 Techniques and Measures

6.2 SFM Validation (3d)

6.2.1 Task

The formalisation of the requirements in form of a semi-formal model enables a systematic check of the completeness and consistency of the system test specification.

The model itself can perhaps be animated (depending on the concrete form which is not yet fixed). This offers the chance to an early (preliminary) validation of the design.

6.2.2 Documents to Be Produced

1. Revised System Test Specification
2. SFM validation report

6.2.3 Phase Specific Activities

6.2.4 Techniques and Measures

6.3 Final Validation (tbd)

6.3.1 Task

The final validation shall ascertain that the end result of the development—the EVC software in its specified environment—behaves as required.

6.3.2 Documents to Be Produced

1. System Test Definition (based on System Test Specification)
2. System Validation Report

6.3.3 Phase Specific Activities

Testing the software against the user requirements.

6.3.4 Techniques and Measures

A technique to be considered is testing in a validated testbed (including API animation/simulation).

Part II

Verification & Validation Plan for the Project openETCS

7 Verification & Validation Strategy for the Project openETCS

Scope of Verification & Validation in openETCS

The project will only perform part of the development, and thus also only a part of the V&V activities. These need to be defined and planned. The overall approach shall be to try out the different constituents on and in representative samples, so that the realisability of the verification & validation strategy for a full development can covincingly be demonstrated.

Besides the usual purpose of verification & validation activities, namely evaluating and proving the suitability of design artifacts, V&V in openETCS will also generate information on the suitability of the methods and tools employed. For that purpose, a format for describing methods and tools to be used in V&V and one for summarizing the findings about the suitability are defined.

Implementing SCRUM

The overall project approach is to organise activities in an agile way, following the ideas behind the SCRUM development method. This has two main aspects for V& V (in the following text, *verification* is used to also include validation):

1. Artifacts are produced in iterations and expected to be verified iteratively.
2. Verification activities are split over several iteration, where each should produce some useful result.

Adressing (1), it is recommended to not attempt to derive a final verdict, except in cases where a set of requirements are claimed to be covered completely. Preliminary models or code parts can serve to evaluate tools and methods, and to set up verification environments, to be able to complete the verification when the artifact is a more final state. Preliminary artifacts should be assessed for their aptness for the intended v&v. Feedback on deficiencies will help to accelerate later activities.

Concerning (2), an adequate procedure is to include a review of the verification in each iteration. This corresponds to the principle of only permit “tested code” into the results of each SCRUM phase. This review should best be performed by a project partner different from the one who performed the verification.

Further practices to do verification during an ongoing development with several parameter not set, include:

1. Write an object of verification if necessary for evaluation purposes
2. Construct specifications from available material

3. Try to fit those into a general picture of the development (process)
4. Select methods and tools prudently (keep FLOSS and CENELEC in mind)

Cooperation with Other WPs

The verification & validation has to be performed in cooperation with WP 3, which produces DAS2Vs (models and code), and with WP 7, where methods and tools are defined and developed.

To exchange information with WP 3, formats are needed for collecting information about DAS2Vs (V&V tasks) and for giving back information about the results of V&V activities. Similarly, with WP 7 communication shall use formats to describe V&V methods and tools (input from WP 7) and the results of evaluations of V&V methods and tools.

8 Verification Plan for the Project openETCS

8.1 Verification Overview

Describe the organization, schedule, resources, responsibilities, tools, techniques, and methodologies to be deployed in order to perform the verification activities.

8.1.1 Organisation

Define the relationship of verification to other efforts such as development, project management, quality assurance, and configuration management. Define the lines of communication within the verification effort, the authority for resolving issues, and the authority for approving verification deliverables.

Organisation: a format for describing design artifacts subject to V&V, and a feedback format for the findings during V&V.

8.1.2 Schedule

The schedule summarizes the various verification tasks and their relationship to the overall openETCS project. It describes the project life cycle and project milestones including completion dates. Summarize the schedule of verification tasks and how verification results provide feedback to the whole openETCS process to support overall project management functions. The objective of this section is to define an orderly flow of material between project activities and verification tasks.

According to the Description of Work of WP 4 [2], the verification activities will be structured into three main phases:

1. First Level: Verification of prototypical system and API model and prototypical code,
2. Second Level: Verification of system model, functional API prototype model, code architecture and system API prototype
3. Third Level: Verification of final system and API model, final code and the functional API model

8.2 Verification Activities—User Stories

The term “User Story” as used in openETCS stands for any kind application of tools, not just for the end user application of the system (EVC software) which is to be developed. This section shall describe such “user stories” of verifiers, i.e., it shall describe where which method or tool is to going to be applied to what artifact(s) (DAS2Vs). It thus shall tell coherently the **story** of verification activities. Later (sub)sections provide the organisational detail: **When** (Timeline, Sec. 8.3) and **Contribution** (which of the verification obligations from Sec. 5.3 are tackled by what approach.

the template for a user story should be included. It needs to be updated: Verification Levels 1 and 2 should be covered. One may include the old plan for Level 1, and mark changes appropriately.

8.2.1 Reviews and Inspections

Reviews

In the openETCS project all written documents, specifications, models and code can be reviewed. It is also important to include all documents concerned with the creation and delivery of the openETCS product. This means that strategies, plans, approaches, operation and maintenance manual, user guides, the contract that will initiate the work should all be reviewed in a structured way.

Information about the reviews planned within openETCS is given in the QA plan [1].

Inspections

Inspection shall be applied to design artifacts whose correctness is important for the demonstration of the project results, and also to validate results of innovative verification & validation methods and/or tools.

8.2.2 Software Architecture Analysis Method (SAAM)

Potential targets for SAAM are parts and instances the SFM (semi-formal model) of the software, to ascertain the viability of the software architecture decision. The scenarios with which the analysis shall be performed are to be developed from the operator requirements.

8.2.3 Architecture Tradeoff Analysis Method (ATAM)

ATAM may be applied to the SW architecture.

8.2.4 Formal Verification at Software Level

This section describes CEA LIST's and Fraunhofer FOKUS' plans regarding the use of formal methods to assess properties at the C code level. Section 18.1 above describes the main plug-ins of the Frama-C tool suite that are envisaged for that, while the theoretical background is summarized in sections 16.1 and 16.2. Namely, two main categories of properties can be dealt with. First, we can focus on functional properties, that is establishing that a given function is conforming to its (formal) specification. Second, it is also possible to analyze a whole application to check the absence of potential run-time errors (arithmetic overflows, division by 0, invalid dereference of pointers, buffer overflow, use of uninitialized variables, undefined order of evaluation, ...). A case study partly based on previous experiments is developed further in OpenETCS and has been presented in [3]. Existing code from OpenETCS partners, namely Siemens and ERSA has also been identified has a good target for such activities.

8.2.5 Applying RT-Tester to a Model of a Component Handling the Acknowledgment of a Level Transition Order

This section describes the verification activity of the DLR during Verification Level 2. The object of verification is a model of a component handling the acknowledgment of a level transition order by the driver. This component shall trigger braking if the driver does not acknowledge the level change as required. The goal of the activity is mainly to evaluate the RT Tester tool as a component of the secondary tool chain. The verification object itself will be created as part of the activity. It is intended to contribute it to WP 3 (Modelling) as a byproduct of this activity.

Object of verification

The object of verification will be available in the modeling repository on GitHub after it has reached a certain degree of maturity. It is intended to constitute a *module design specification*, i.e., a specification to be implemented using SCADE to generate code in a subsequent development step. It will be developed as a SysML model with C annotations to make it executable. It describes how the OBU monitors potentially necessary driver acknowledgments after a `Level Transition Order` (LTO) has been received. The term LTO Monitor will be used to refer to the component in the ensuing description.

Available specification

Section 5.10.4 of Subset 026 is taken as functional specification. Details like the interface objects of the LTO Monitor and functionalities of other OBU components which the monitor relies on will be defined based on a general understanding of the workings of the OBU and a discussion with project contributors (lacking a SW architecture and component design so far).

Methods and Means

The main tools employed in this activity are Papyrus for SysML modeling (Primary toolchain, deliverable D7.1) and RT Tester, Sec. 14.6. For modeling, we adhere to the language restrictions of the RT-Tester manual [*add citation*] in order to be able to use the model as a *simulation*. This way, we can compile the LTO Monitor and run tests on it. Tests are to be generated from *test models* focusing on parts of the monitor functionality.

Results to be achieved

The intention is get a feeling for the capabilities of RT-Tester to be able to assess its potential for implementing the openETCS workflow. Findings and conclusions shall be entered into a report, to be discussed with U Bremen (who contribute the RT Tester) and potential users of it.

Timeline

- Modeling the LTO Monitor, writing test models and applying the tests to the monitor shall be done during the second level of verification. The activity shall be completed by October 2014.
- There are no results so far as the activity has started with Verification Level 2.
- No plans for continuing the activity in Verification Level 3 have been made.

Maturity Classification

The tools applied have the following TRLs (Technology Readiness Levels):

Papyrus: TRL 4 (tentative estimation). Papyrus SysML modeling has been employed by the developers and potentially also in an industrial context, but not in the form it is done here. WP 7 will be in a better position to judge that.

RT Tester: TRL between 4 and 8. According to U Bremen (Jan Peleska), RT Tester has been assessed for use in the development regulated by relevant standards. More details on that will be provided by U Bremen. Wrt. the current usage context (Papyrus SysML), the evidence would have to be evaluated to assign a similar TRL (higher than 4).

This activity shall not comply to the requirements of a SIL 4 development, as it is intended to evaluate the tools. A positive outcome of the evaluation would contribute to an argument that RT Tester is appropriate to perform a part of the verification of design specifications which take the form of executable models. This relies on:

1. A potentially high maturity of RT Tester (its qualification)
2. The fact that systematic dynamic testing with high coverage is an element of an approved combination of techniques for code verification. And that executable models would be adequately treated similar to code.

This is of course a preliminary statement to be substantiated at the end of the activity.

8.2.6 Formal Model Verification (TWT)

This section describes TWT's plans regarding the formal verification of the system model. While TWT's original approach has been focused primarily on real-time aspects by employing timed automata and the UPPAAL tool as described in Section 18.6 (page 76) our activities are now addressing formal model verification on a broader scale.

TWT's work will be based on the following action items:

1. We will investigate the suitability of the open source toolkit CPN Tools[4] for formal verfication (model checking) and simulation. The evaluation of CPN Tools will be aligned with the secondary toolchain assessment in WP7. A first step in the tool evaluation is the construction of an example model (currently "Start of Mission") from the ETCS specification.
2. We will analyze whether/how UML/SysML statecharts and possibly activity diagrams can be transformed to colored Petri nets to permit their simulation and the formal verification of requirements.
3. We will analyse the expressive power of the CPN Tools property language and whether the modeling formalism can replace UPPAAL for modelling and verifying timed aspects of ETCS components.
4. We will provide feedback regarding ambiguities, inconsistencies and errors in the current ETCS standard based on our formalisation and the analysis of other models based on our approach.
5. As the University of Braunschweig has experience with formal Petri net modelling, we will align our efforts with them.

6. We will investigate, how our approach can be integrated in the Eclipse environment and develop such integration if feasible with our efforts.
7. We plan to publish our results on action item 1.

Please note that the above list may be subject to future change.

8.2.7 Verification with Model-Based Simulation using SystemC (TWT, URO)

In Section 19 (page 79) the basics of SystemC and the SysML/SystemC joint approach have been described. TWT and the University of Rostock (URO) will work on this concept based on the following action items:

1. TWT will analyse methods for generating SystemC code from SysML models. In first investigations the Acceleo tool (Eclipse plugin, based on OMG standard) seems to be a promising candidate for model-to-text transformation.
2. URO will build a modular and executable SystemC model for braking curves that is suitable for real-time simulation. An accompanying high-level SysML model will be constructed as well and will be means to test the transformation methods to be developed in the context of action item 1.
3. TWT and URO plan to investigate which other parts of the ETCS specification can benefit from real-time simulation and build models accordingly.
4. URO will investigate whether performance analysis based on the underlying hardware system is feasible within the openETCS project. This will allow to scale the hardware resources of the OBU system accordingly.
5. In addition, using the results from action item 1, TWT and URO plan to transform existing SysML models (to be developed in WP3) to SystemC for real-time simulation.
6. Evaluation (and possibly implementation) of an Eclipse integration

Please note that the above list may be subject to future change.

8.2.8 System Integration Testing (Uni Bremen/DLR)

Section 14 describes the main objective of system integration testing. Uni Bremen will focus on the following items :

1. Create a test model in SysML. From the model evaluation activities, the management of the radio communication is already available. To cover different aspect of the specification, the ceiling speed monitoring model will be also provided.
2. Generate test cases according to the defined interface given by DLR. (RT-Tester)
3. Provide simulation environment for the track-to-train simulation (including braking curves/speed profiles)[2] along routes used for testing

[2] OpenETCS system testing for the EVC on-board computer requires test execution in real physical time and also track layout with realistic speed profiles. We also want to contribute to the track and Speed Simulation by automatically generates “relevant” layout for OpenETCS.

4. Study the automatic generation of these track layouts and speed simulations; if feasible, implement a generator and integrated it in the DLR laboratory environment
5. Set up a test environments for
   - Hardware-in-the-loop Testing within DLR laboratory.
   - Software-in-loop testing with code provided by SCADE (Siemens)

Track simulation

OpenETCS system testing for the EVC on-board computer requires test execution in real physical time and also track layout with realistic speed profiles. We also want to contribute to the track and Speed Simulation by automatically generates “relevant” layout for OpenETCS.

Test cases generation

We also plan different activities to ensure the pertinence of our test cases.

1. Check the test model (SysML) : RTT-BMC or other model checkers
2. Add relevant LTL properties if needed
3. Test case analysis by
   - Structural coverage
   - Requirement coverage
   - Mutation coverage
   - Data coverage

Test cases analysis – comparison to Subset 76

1. Provide techniques and Howto describing how test cases from Subset 76 can be executed in the RT-Tester environment, either as SW integration test or as HiL test in the DLR simulation environment
2. Create new set of test cases for the ceiling speed monitoring (As far as we know,they do not yet exist in Subset 76)
3. Compare new test cases created by RT-Tester to new test cases for ceiling speed monitoring provided by ERTMS standardization group, as soon as available; suggest improvements for the Subset 76 test cases.

Exchange Formats

Test models represented in XMI/Ecore are used as SysML test modeling standard. RT-Tester model parsers are extended to cope with this format.

Test procedures will be represented in a general abstract syntax format, so that procedures generated by RT-tester can be run on any test execution platform.

Test results (test execution logs) will be represented in a general format, so that exchange of test results between tools (for example, for simulation purposes) becomes possible.

A first part of these activities have already been done for model of radio management communication (see the model-evaluation for EA/RT-tester).

8.2.9 Model Verification by applying the openETCS Verification Tool Chain (Siemens)

Siemens will focus the activities on the verification of its contributions to the openETCS work packages 3 and 7. Currently available is the MoRC model provided for the primary tool chain for WP 7. This modeling will be continued for WP 3. The existing formal model will be complemented by a semi-formal model on top. Both, the semi-formal as well as the formal model need to be verified in WP 4 and will serve as verification objects.

The intention is to apply the preferred and most valuable methods and parts of the openETCS verification tool chain to the models and by giving feedback into the openETCS project contribute to increase their useability and adequacy.

In more detail, these activities are planned:

- Support the integration of the MoRC model into the openETCS test environment / DLR laboratory.
- Support the integration of the MoRC model into the RT-Tester environment, provided by Uni Bremen.
- Model based test case generation and execution by using the RT-Tester environment provided by Uni Bremen.
- Determination of the structural test coverage on the model.
- Determination of the requirements coverage regarding subset-026 documents, SSRS, implementation model, test model, test case and test execution issues.
- Determine the feasibility of proving safety properties for the model.

8.3 Verification Activities—Timeline

This section lists per partner/activity in which of the project phases according to Sec. 8.3 (first, second or third level of verification) a particular activity is planned. In the first version of the V&V plan, only the first level needs to be detailed.

8.3.1 First Level of Verification

This section gives only a short description, which should refer to an activity detailed earlier. Most probably the material will be organised in a table.

TWT

TWT will continue with the analysis of methods for generating SystemC code from SysML models. In first investigations the Acceleo tool (Eclipse plugin, based on OMG standard) seems

to be a promising candidate for model-to-text transformation. The approach will be aligned with URO and their SystemC model of the braking curves. See also Sct. 8.2.7.

In addition, TWT will start with the analysis on whether/how UML/SysML statecharts can be transformed to timed automata in a sensible manner while retaining as much structural information as possible. Timed automata provide a means for model checking real-time properties of systems. See also Sct. 8.2.6.

URO

URO will continue their work on a modular and executable SystemC model for braking curves suitable for real-time simulation. An accompanying high-level SysML model will be constructed as well and aligned with TWT's activities on SysML → SystemC code generation. See also Sct. 8.2.7.

Uni. Bremen

Uni Bremen will start by developing and setting a simulation environment for the test of the EVC. The test models from the model-evaluation will be completed and the breaking curves will be added. The set of tests generated will be then compared to the available ones of SUBSET-076.

8.3.2 Second Level of Verification

8.3.3 Third Level of Verification

8.4 Verification Activities—Process View

This section provides the detailed plan for the verification tasks throughout the openETCS project life cycle. It summarizes the activities performed by the project partners in relating them to the overall definition of verification activities in Sec. 5.3.

8.5 Verification Reporting

This section describes how the results of implementing the Verification Plan will be documented. Verification reporting will occur throughout the software life cycle. The content, format, and timing of all verification reports shall be specified in this section.

Here we need the template for reporting.

The following reports will be generated during the verification process:

- **Anomaly reports:**
- **Phase Summary Verification reports:**
- **Final report:**

The structure of the Verification report is already defined in the 5.4.1 section of this document

8.6 Administrative procedures

This section identifies the existing administrative procedures that are to be implemented as part of the Verification Plan. Verification efforts consist of both management and technical tasks. Furthermore, it is the task of the SQA team to monitor whether the procedures as defined in the management plans ([QAplan], [SCMP], [Review and Revision processes]) are followed.

8.6.1 Problem Report

The problem reporting procedure is described within the document Change/Problem Management Process.

Any problem, failure and error encountered during the review activities (QA. Verification, Validation, Assessment) planned in the software development life-cycle, problems reported by users and customers as well as change requests initiated by any of the system stakeholders will be reported and managed following the Change/Problem Management Process detailed in [governance] and through the Change/Problem Management Tool.

8.6.2 Task Iteration Process

Any change in the requirements (system, sub-systems, sw or components) require repeated verification and validation activities.

Once the change is accepted following the change/problem management procedure, the phases and items affected by it must be evaluated. These tests will be redesigned to reflect the change in the requirement and will be executed again.

In turn, a new analysis of the Software Integrity Level will involve the analysis of the activities requirements and documentation presented by the EN50128 standard and include such activities in the SVVP if necessary.

8.6.3 Deviation Process

The Quality Manager will be informed in the case of detection of a deviation regarding Verification Plan. In addition, he/she also be informed if it is deemed necessary by an amendment to the Plan, whether or not motivated by a deviation

The Quality Manager will report such incidents to the Project Managers and with whom shall act appropriately. All persons listed in the Responsibilities section (Sec. 5.1.5) shall be informed of a change in the Verification Plan

8.6.4 Control Procedure

Control procedures are specified in the Configuration Management Plan [SCMP]

9 Validation Plan for the Project openETCS

9.1 Validation Overview

This section describes the validation activities which are planned to be performed within the project.

The timeline of validation activities parallels that of the verfication. I.e., there are the same three levels as in Sec. 8.1.

9.2 Validation Activities—User Sories

Like Sec. 8.2, this section describes the validation activities to be performed within the project. It shall tell coherently the **content** (method, tool, result quality) of validation activities. Later (sub)sections provide the organisational detail: **When** (Timeline, Sec. 9.3) and **Contribution** (which of the validation obligations from Sec. 6 are tackled by what approach.

9.3 Validation Activities—Timeline

This section lists per partner/activity in which of the project phases according to Sec. 9.3 (first, second or third level of validation) a particular activity is planned. In the first version of the V&V plan, only the first level needs to be detailed.

9.3.1 First Level of Validation

A short description which may refer to an activity detailed earlier. Most probably the material will be organised in a table.

9.3.2 Second Level of Validation

9.3.3 Third Level of Validation

9.4 Validation Activities—Process View

This section provides the detailed plan for the validation tasks throughout the openETCS project life cycle. It summarizes the activities performed by the project partners in relating them to the overall definition of validation activities in Sec. 6.

Part III

Methods and Tools for Verification and Validation

The project shall select / develop / describe a chain of methods and tools for doing verification & validation in a full development. In Version V01, this section collects proposals, which will be evaluated and from which those suitable for real-life development and maintenance of the open-source EVC software will be selected.

Each proposal described here shall be classified for which tasks and activities in teh development it is intended, i.e., which role it should take.

A common format should be included, adressing:

Contributor Person or organisation who contributed the description. Should name a contact for further information concerning the method or tool. In case of modifications or additions by others than the contributing party, these others shall be added.

Purpose For which activites this method/tools might be useful

Evaluation/Evidence Has it been demonstrated somehow? Summary, pointers, etc.

Review Has the evidence been assessed (internal asssessment, verdict)?

Tool qualification Needs/done/Ideas

10 On the Notion of “Formal Methods”

As FLOSS relies to a large extent on the use of tools for generating, verifying and validating a design, “formal methods” necessarily will play an important role in the openETCS activities.

In common language, the notion *“formal”* is often used in a broad sense, meaning everything that can be described by rules, even if they are rather vague. Contrary to that, we use *“formal”* in the narrow sense of EN-50128 [5, Section D.28], meaning strictly mathematical techniques and methods. Since the Aerospace Standard DO-178C [6] follows a similar understanding, but gives more elaborate explanation in its supplementary document devoted to formal methods [7], our presentation closely follows the terminology of the latter.

> *Formal methods are mathematically based techniques for the specification, development, and verification of software aspects of digital systems. The mathematical basis of formal methods consists of formal logic, discrete mathematics, and computer-readable languages. The use of formal methods is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analyses can contribute to establishing the correctness and robustness of a design.*
>
> [7, Section 1.0, p.1]

11 Reviews and Inspections

Not everything can be done by formal methods. The introduction of reviews in the openETCS project will enable the adherence to design specifications as well as ensuring that appropriate coding techniques—for automatic generation as well as observation of relevant coding standards—have been used. The use of an appropriate review technique will also help to ensure the consistency of approaches across development teams and may result in improved standards through the identification of best practice solutions. Software reliability will be increased due to the removal of a larger percentage of the errors that would otherwise remain in the software. Specific analysis techniques exist for predicting the reliability of software.

For verifying any product, whether it is a piece of software, a design specification, test script or anything else, it is essential to carry out some sort of review or inspection process. The introduction of formal review processes provides us with definite points in time when we can carry out these essential assessments.

It is possible to review just about anything. In the openETCS project all written documents, specifications, models and code can be reviewed. It is also important to include all documents concerned with the creation and delivery of the openETCS product. This means that strategies, plans, approaches, operation and maintenance manual, user guides, the contract that will initiate the work should all be reviewed in a structured way.

Inspection

On the other hand, an inspection is a visual examination of a software product to detect and identify software anomalies, including errors and deviations from standards and specifications [8]. Inspections are peer examinations led by impartial facilitators who are trained in inspection techniques. Determination of remedial or investigative action for an anomaly is a mandatory element of a software inspection, although the solution should not be determined in the inspection meeting.

Both review and inspection methods shall be used in different contexts during the execution of the Verification activities.

More information about the reviews can be found in Quality Assurance Plan [1].

12 Software Architecture Analysis Method (SAAM)

SAAM [9] is one of the simpler methods for a scenario-based architecture evaluation, and it was the first to be published. SAAM is suitable for the testing of software architectures with regard to quality attributes (qualitative requirements), such as

- Modifiability,
- Portability,
- Growth Potential,
- Performance,
- Reliability,

but also for the evaluation of the functionality (functional requirements) of a software architecture.

In a SAAM evaluation basically scenarios are developed, prioritized and assigned to those parts of the software architecture to be tested that are affected by them. This may be sufficient to indicate problems in the architecture.

13 Architecture Tradeoff Analysis Method (ATAM)

ATAM [10] is used to review the design decisions of the architecture. It is checked whether the design decisions satisfactorily support the requirements concerning quality. Risks and compromises included in the architecture are identified and documented.

The process includes two phases.

- In the first phase the necessary components are presented. Then the architecture is checked and analyzed.
- In the second phase it is tested whether the analysis and the test were correct and complete. Then the results are summed up.

14 Model Based Testing Method

14.1 Model Based Testing Strategy - generalities

Testing consists in executing the System Under Test (SUT) for some particular inputs and in assessing whether or not the corresponding SUT executions conform to some requirements. Whatever the testing technique used is, one has to define test cases to be submitted to the SUT and associate to them a decision procedure called oracle. The oracle allows the tester to compute verdicts according to what the executions of SUT (resulting from the test case submission) reveal about its correctness. This correctness is measured with respect to requirements. Model based testing is a particular kind of testing technique in which requirements are described by models which are executable specifications. Their execution traces (or "traces" for short) are sequences of stimulations of the SUT and resulting observations of the SUT reactions. Test cases are sequences of stimulations that are selected from the test model. A sequence corresponding to an input test data can be obtained by considering a trace of the model and "forgetting" it. For functional testing, SUT is considered as a black bbox: the tester (a human or a test bench) can only stimulate the SUT and observe its reactions. Interactions between the tester and the SUT result on the definition of traces. Therefore, a SUT can be seen as a set of traces that is not known (since SUT is a black box) but the tester may discover some of those traces by interacting with SUT. The oracle is based on a so called conformance relation. A conformance relation is a mathematical relation between the set of traces of the SUT and the set of traces of the model. When these sets of traces fulfill the relation we say that the SUT conforms to the model. The oracle takes as inputs traces representing an interaction between the tester and the SUT and compute verdicts. Whatever the testing technique is, the set of possible verdicts always contain the verdict Fail which is emitted whenever the trace taken as input demonstrates that the SUT does not conform to the model. Depending on the testing technique used there may be different verdicts emitted when Fail is not emitted. These different verdicts reflect different traceability information related to interaction trace taken as input. In this section we briefly discuss two model based testing tools that we will use conjointly in the OpenETCS project.

Model based testing (MBT) may apply at different level during the lifecycle:

- System integration testing
- Software integration testing
- High level code verification
- Object code verification

High-level code verification may be performed on any host perform whereas object code verification intends to test the running code on the target hardware.

Model-Based System Integration Testing

The objectives of MBT on system integration level are to

- validate the correctness and completeness of the development model,

- verify that the generated code components cooperate correctly on the target HW, in order to achieve the system-level capabilities.

The first objective implies that the *test model* and the original development model are separate entities; otherwise the system integration test would just validate that all logical errors still residing in the openETCS development model are really implemented in the code. Even in presence of a formally validated development model, in which high confidence can be placed, we prefer to create a separate test model, because

- the test model may use a higher level of abstraction since only the SUT behaviour visible at the system interfaces is relevant,
- the test model may specify different interfaces to the SUT, depending on the observable interfaces in a test suite; the observation level ranges from black-box (only the “real” SUT system interfaces are visible) to grey-box level (some global variables may be monitored or even manipulated by the testing environment, some task or object communications may be observed etc.),
- the development model may contain errors that are only revealed during HW/SW integration (for example, calculations failing due to inadequate register word size, or deadlines missed due to insufficient CPU resources).

Software integration is performed by software-in-the-loop technology on host computers. The software components as well as the complete software are tested on host computer with a testing environment that simulates the hardware behavior and the operational environment. The main advantage is that all software properties can be easily simulated and tested. The tests for software-in-the-loop may be generated from a model but at this level unite test for each software functionalities may also be performed.

Model-Based Testing of Generated High-Level Code

Another application of MBT aims at the verification of generated high-level code (for openETCS, the target language will be C). If model-to-text transformations are not formally verified, it is necessary to verify the outcome of each transformation. Since the transformation source is a model M, MBT suites can be derived automatically from this model to show that the generated code conforms to M.

Observe that in contrast to system-level MBT no redundant model is used for this objective, but the same model M used for code generation can be used: we just have to verify the consistency between code and M, without validating M’s correctness and completeness. The latter task is separately performed by means of

- property checking or
- simulation.

The model-based testing (MBT) approach can be used to create test suites conforming to the highest criticality level of the applicable CENELEC standards, in order to justify that the generated code is consistent to its model [11, 12, 13]. Furthermore, the generated result may

be formally verified against the model. This formal verification task is easier than proving the correctness of a generator or compiler as a whole, because now just one concrete artefact (the generated code) has to be checked against the transformation source. The theoretical foundations of object code verification, as well as its proof of concept have been established in [14]. In [15, 16] these concepts have been refined and applied to the railway domain.

The main advantage of this approach in comparison to performing V&V for generators and compilers is that the latter do not have to be re-verified after improvements and extensions. Therefore we advocate the test-based code verification approach to be applied in openETCS for verifying generated high-level source code or object code of SIL-4 applications.

The HW/SW integration testing is out of the scope of this project. Nevertheless, Model-Based Testing of Compiled Object Code or/and Alternative Unit test on target HW may also be performed.

14.2 Model Based Testing applied to Open ETCS V&V

According to the previous activity on defining the project process, the Open ETCS process is based on 3 main inputs for methodology and product lifecycle: the SCRUM methodology, the Model Driven Design and the Cenelec software development V cycle. Traditionally, system requirements are directly translated into formal specifications on which verification and proof techniques are applied. The use of formal specifications and formal language allows then to derive the models using dedicated languages (B for instance) in order to guaranty conservation of properties along the design process. The main difficulty in this context is to be sure that the interpretation of rules has correctly been captured in the formalized specification which is not easy to check by the regulators. For this reason, the model based testing has been chosen as testing and verification technique within the V&V activities of the OpenETCS project.

Moreover, we suggest to create test models on the basis of the ETCS standard (subset 026) and the existing high-level test suites made available in subset 076. The latter test cases should be feasible computations of the test model, so that the test model really creates a *superset* of the existing test suite from subset 076.

This technique application will be explained in the following paragraphs through the description of different model-based testing tools: MaTeLo, Diversity and RT-tester.

14.3 Matelo Model Based Testing solution

MaTeLo purpose is to generate test cases for systems whose expected usage and behavior are described by a probabilistic model. MaTeLo tool is based on its own test model called "usage model" and uses, among other characteristics, usage profiles for test case generation. This usage model describes the possibilities regarding the use of the soft (in our case; operating scenario) during its whole lifecycle. This usage model is performed thanks to a Matelo specific modeler, it allows to generate test cases that will then be plugged to the SUT which will be the software semi-formal model realized in the frame of WP3 activities. MaTeLo has three main functionalities: test modeler, test cases generator and test campaign analyser. Even if MaTeLo is mainly a test case generation tool, we can consider that this tool performs also analysis for different reasons:

- A test model can be considered as a development artifact the same way as a system model for example. So analysis on it could identify some ambiguous or erroneous points in test model

(i.e. in the future test campaign) or in the specifications (because MaTeLo mode l is built from system specifications).

- Even whether test campaign analysis is mainly based on testing activities, analysis techniques have to be used as wellThe limit between a model to perform test and a model to perform analysis is not so obvious.

Because its test case generation is based on a model, MaTeLo belongs to the family of Model-Based Testing solutions. MaTeLo model basically uses Markov Chains to describe the test model of the SUT implemented for "Black Box Testing" in all xIL steps (MIL, SIL, PIL, HIL). MaTeLo Usage Model edition facility allows for implementing test models that describe the use cases of the SUT completed with the tester point of view, and then, Matelo testing facility can generate automatically the test cases generated by the tool. Thanks to the numerous validation steps, MaTeLo Test Campaign Analysis provides information such as test coverage (requirements, model) or reliability of the SUT. Once the MaTeLo test model is performed and the testing strategy is defined with MaTeLo profiles faciulities, MaTeLo generates test cases. For that, MaTeLo Testor contains several test generation algorithms that can be used for different purposes. Different test case generators are based on a Usage profile approach, considering the occurrence probability of each model transition. Other are deterministic (most probable execution path, or all the transitions are covered). In the case of Open ETCS project, the SUT model is an on-board EVC, designed according to the SRS Subset 026. This specification itself is not sufficient to cover all functional aspects, and tests depend strongly on the operating rules to be considered on the observed track. The principle for the MaTeLo model would be to encompass all the possible states and transitions that can be considered in a well-defined perimeter (based on Subset026, signaling and exploitation rules to consider). Then, the test could be precisely defined by the usage profile to adapt it to a track oriented testing campaign.

14.4 Diversity Model Based Testing solution

DIVERSITY is a model based testing tool developed at CEA LIST. Its underlying technology is symbolic execution. Symbolic execution has been first defined for programs. The goal of this technique is to identify, for each possible execution of the program, the constraints to be satisfied in order to follow it. The main idea consists in executing the program, not for concrete numerical values but for symbolic parameters, and to characterize constraints on these parameters at each step of the execution. In that sens, DIVERSITY is a white box testing tool. In the frame of the openETCS project we plan to use DIVERSITY to extract test cases from models defined in the first phases of the system design. Our goal is to extract test cases dedicated to abstract safety requirements. More precisely we focus on safety requirements dealing with communication between sub systems. For that purpose we will use the language of sequence diagrams extended with timing constraints to specify such requirements. With sequence diagrams, one may describe execution scenarios in terms of partially ordered message passing between subsystems. Message passing can be structured thanks to operators expressing sequencing, parallelism, choice, loop... It is possible to automatically analyze sequence diagrams with DIVERSITY in order to extract test cases. The originality is that, thanks to projection mechanisms, it is possible to extract test cases, not only for the entire system, but also for any sub systems composing it. Because of this mechanism, sub systems can be tested as soon as they are implemented, even though the entire system is not yet implemented. In such a process we perform a particular kind of unitary testing in which unit test cases are built according to the usage that will be made of the sub system in the entire system. In the frame of OpenETCS, this functionality could be useful in order to realize the unitary and modular tests. The first step consists in defining a requirement model in the form of a sequence diagram or a Matelo Test scenario. The requirement model is

analyzed with DIVERSITY in step 2. This analysis results on a so-called symbolic tree, whose each path denotes a possible (symbolic) execution of the sequence diagram. Such trees may be theoretically infinite due to the possible occurrences of the “loop” operator of sequence diagrams. Therefore, DIVERSITY uses various stopping criteria to stop the computation (typically based on message coverage notions). The symbolic tree computed in step 2 characterizes executions of the whole system model. However, because testing the whole system may be complicated in terms of testing architecture, or simply because one wants to test some sub systems before the whole system is implemented, we offer a mechanism to extract symbolic trees for each distinguished sub system. This is based on so-called projection techniques. This operation is realized in step 3. In step 4, each identified sub system is tested thanks to a real time off-line testing algorithm. Then, we can relate correctness of sub systems and correctness of the whole system by using a compositionality theorem. The compositionality theorem expresses that, the conformance of each subsystems to all their projections guarantees the conformance of the whole system to the sequence diagram. A direct consequence is that any faults of the whole system can be discovered as a fault of at least one of its sub systems. This implies that testing the whole system mainly comes to test each of its sub systems after a short test integration phase testing that each sub system is correctly connected. We believe that such an approach will be very useful for ETCS systems which are by nature very distributed and thus hardly observable and controllable as a whole. The share of OBU EVC kernel in sub-system is the role of the SSRS model, and this refinement to diversity will be possible once this functional decomposition of the EVC will be released.

14.5 Complementary use of the DIVERSITY and MaTeLo

The use of the two tools can be done in a complementary way that would allow a more efficient test case set generation. MaTeLo would start from the test model, and generate automatically all the use cases that can be encountered in CBTC use. MaTeLo tool analyses the models as black box, and generates tests according to a stochastic approach. DIVERSITY will analyze these scenarios, based on a symbolic execution of the semi-formal SysML model (white box testing), in order to filter the tests generated by MaTeLo and to reduce the test case set. As discussed in previous Sections, the two tools DIVERSITY and MaTeLo handle different kinds of models. The version of DIVERSITY that we will use in the project handles high level models in the form of sequence diagrams. Such models can be used to specify requirements on communication scenario between subsystems of a reference system under test. Models handled in MaTeLo are automata labeled by transfer functions and probabilities. Such models are useful to describe executable behaviors very close to the actual implementation, and based on operating scenarii. Clearly these two levels of modeling are useful in design processes of safety critical applications such as ETCS implementations, and can be combined in different ways for improving the test coverage of our EVC Software kernel. Indeed, ETCS systems have such a level of complexity, that it is difficult to describe them in a model straight from the requirements. Therefore, the refinements provided by two modeling levels are very helpful. Moreover, it is mandatory to maintain a good traceability between these two levels of modeling, in order to fulfill the safety requirements. The complementarity of these tools takes place in some refinement processes in which high level requirements can be implemented into executable models. However, it is crucial to assess whether executable models correctly implement requirements. In practice this may be a difficult question because it requires to efficiently explore the executable model, which by nature is generally huge because it represents in a precise manner the functional behaviors of the actual implementation. In order to overcome this problem we plan to take benefits from the fact that executable models of ETCS will be described in the form of communicating executable models. This fact permits to see the model as a collection of communicating subsystems. This permits to take benefits of the compositional result described in the Diversity, and use it for white box

testing (the internal behavior of functional modules and blocks defined in the kernel can then be precisely tested)..

14.6 The RT-tester

The RT-Tester test automation tool, made by Verified [17], performs automatic test generation, test execution and real-time test evaluation. It supports different testing approach such as unit testing, software integration testing for component, hardware/software integration testing and system integration testing. The RT-Tester version follows the model-based testing approach [18] and it provides the following features :

- Automated Test Case Generation
- Automated Test Data Generation
- Automated Test Procedure Generation
- Automated Requirement Tracing
- Test Management system

Starting from a test model design with UML/SYML, the RT-tester fully automatically generates test cases. They are then specified as test data (sequences of stimuli with timing constraints) and used to stimulate the SUT and run concurently with the generated test oracles. The test procedure is the combination of the test oracles and the SUT that can be compiled and executed.

The tool supports test cases/data generation for structural testing. It automatically generates reach statement coverage, branch coverage and modified condition/decision coverage (MC/DC) as far as this is possible. The test cases may all be linked to requirements ensuring a complete requirement traceability. Additionally RT-tester may produce test cases/data from a LTL formula, since a LTL formula describes a possible run of the model.

Taking advantage of SysML requirements diagram, the test cases and test procedures are directly linked to the requirements. It is then possible to perform test campaign guided by requirements.

Finally the tool may produce the documentation of tests for certification purposes. For each test cases the following document are produced :

- *Test procedure*: that specifies how one test case can be executed, its associated test data produced and how the SUT reactions are evaluated against the expected results.
- *Test report*: that summarizes all relevant information about the test execution.

In [19], a general approach on how to qualify model-based testing tool according to the standard ISO 26262 ad RTCA DO178C has been proposed and applied with success to the RT-tester tool. Following the same approach compatibility with the CENELEC EN50128 may be easily done.

15 Characterisation of Formal Methods

Based on rigorous mathematical notions, formal methods may be used to describe software systems' requirements in an unambiguous way, thus supporting precise communication between engineers. Formally specified requirements can be checked for consistency and completeness by appropriate tools; also, compliance between different representation levels of specification can be verified. Formal methods allow one to check software properties like:

- Freedom from exceptions
- Freedom from deadlock
- Non-interference between different levels of criticality
- Worst case resource usage (execution time, stack, . . .)
- Correct synchronous or asynchronous behaviour, including absence of unintended behaviour
- Absence of run-time error
- Consistency of a formal model
- Correctness of a formal model

In order to subsume this variety of applications under a single paradigm, the DO-178C considers a formal method to consist in applying a formal *analysis* to a formal *model*. Both analysis and model differs depending on the particular method. For most methods, the model is just identical to the source code; however, it may also be e.g. a tool-internally generated abstract state space (used in the Abstract Interpretation method, cf. Section 16.1 below). For most methods, analysis tools need human advice; however, they may also be fully automatic (e.g. for Abstract Interpretation or Model Checking, cf. 16.3).

16 Formal Analysis Methods

In this section we present the three most common methods for formal analysis. The foundation of these analysis methods are well understood and they have been applied to many practical problems.

16.1 Abstract Interpretation

The abstract interpretation method [20] builds at every point of a given program a conservative[3] abstraction of the set of *all* possible states that may occur there during any execution run. Such a representation is also called an *over-approximation*, in the sense that it captures all possible concrete behaviours of the program, while the abstraction might lead to consider states that cannot occur in a concrete execution. Abstract interpretation determines particular effects of the program relevant for the properties to be analysed, but does not actually execute it. This allows one to statically determine dynamic properties of infinite-state programs. The main application is to check the absence of runtime errors, like e.g. dereferencing of null-pointers, zero-divides, and out-of-bound array accesses. While conventional ad-hoc static analysis tools such as PCLint or QAC++ are well-tailored for quick, but incomplete analyses, abstract-interpretation based tools while requiring more computation time, are *safe* in the sense that they guarantee that *all* potential runtime errors are detected. On the other hand, such a tool might report spurious warnings, related to states that are included in the abstraction but do not correspond to concrete executions. Such *false alarms* can be avoided to some extent by increasing the precision of the abstraction [21], at the expense of the computation time of the analysis. However, human intervention is often required to improve the approximation accuracy w.r.t. those program points where *false alarms* have to be removed.

16.2 Deductive Verification

Deductive methods [22] [23][24] [25] perform mathematical proofs to establish formally specified properties of a given program, thus providing rigorous evidence. Its primary use is to verify functional properties of the program. This method is based on the Hoare logic [26, 27], or axiomatic semantics, in which functions are seen as predicate transformers. In summary, a function `f` is given a state described by a given predicate P and transforms it into a new state, described by another predicate $f(P)$. In this context, the specification of `f` is given by a *contract*, which defines the predicate R that `f` requires from its callers and the predicate E that it ensures upon return. Verifying the implementation against such a specification amounts to proving that for each P such that $P \Rightarrow R$ (i.e. that satisfies the requirement of `f`), then $E \Rightarrow f(P)$ (i.e. the concrete final state is implied by what `f` ensures).

Tools based on deductive verification usually extract proof obligations from program code and property specifications and attempt to prove them, either automatically or interactively. Some tools are tightly coupled to a given theorem prover such as Atelier B [28] or Rodin [29], while other such as Why3 [30] promote a cooperation across a wide range of provers. In addition to the contracts of the function, it is often required to provided additional annotations in order to be able to use deductive verification. In particular, for each loop in the code, a suitable *loop invariant* has to be provided. A loop invariant is a property that is true when encountering the loop for the

[3] i.e. guaranteeing soundness

first time and, if true at the beginning of a loop step, stays true at the end of this step. From both hypotheses, it is then possible to inductively conclude that the invariant is true for any number of step, and in particular at the end of the loop. While it is possible to synthesize automatically loop invariant in some simple cases, in particular thanks to abstract interpretation, this activity must most of the time be done manually.

Similarly, some proof obligations are too complicated to be handled by automated theorem provers, and must be discharged interactively via proof assistants [31, 32]. Deductive verification is thus much less automated than abstract interpretation. On the other hand, it is much more flexible for functional properties verification, in the sense that it can be used to prove any property that can be expressed in the specification language of the tool (usually any first-order logic property), while abstract interpretation is limited to the properties that fit within the abstract setting that has been chosen.

16.3 Model Checking

Model checking [33] explores all possible behaviours of a program to determine whether a specified property is satisfied. It is applicable only to programs with reasonable small state spaces; the specifications are usually about temporal properties. If a property is unsatisfied, a counter-example can be generated automatically, showing a use case leading to property violation.

Bounded Model checking [35] (BMC) is mainly used for finding bugs more than proving properties. The basic idea of BMC is to find a counter-example-trace of a bounded length . The transition relation of the system is unrolled to a bounded length symbolically and check against a propositional formula with a SAT or a SMT-solver. If the formula is satisfiable, it exists a feasible path in the system that validate the formula, the SAT solver returns a satisfying assignment that is transformed into a counter-example. Otherwise the bound is increased and the process is repeated. Some more recent works [36, 37] add the use of induction techniques and interpolant to prove the properties.

17 Correct by Construction Formal Methods

17.1 Event B for system analysis

17.2 Classical B for software development

Classical B [24] is used for software development. It is supported by Atelier B from ClearSy[28], a free partially open source tool. The B software development starts with software requirements usually expressed in a natural language document. Classical B uses a mathematical language suitable both to model the requirements and to write implementable code. In a first step called formal general designed, requirements are modeled into B specification modules. The correctness of this step is partially covered by proof and partially by human verification. Then in a second step called formal detailed design, B specification modules are refined by breaking down modules until the B model is fully implemented. The correctness of this step is entirely covered by proof.

A B module contains state data and treatments both abstract and concrete. Those data are based on integers, sets, relations and functions. The properties of data are expressed with classical first order predicates. The treatments, which are called operations, are expressed with substitutions, which may be abstract (for instance one may specify that some state variables become such that some predicate should hold), or concrete like mere statements (assignment, if, case, while).

A B module should follow a refinement process. During this process, specification information is gradually added, design choices are made, and eventually code is written by breaking down the module in other modules. The refinement consistency is handled by dedicated proof obligations, generated by the tool. Those proof obligations should be proved using specific proof tools: some automatic provers try to discharge proof obligations and when they fail, the user should help to build a demonstration in the interactive prover.

During the proof activity of large models, the model should be tuned to build a complete and consistent model where everything should fit together. Through this process, the user becomes aware of the precise properties needed to satisfied every part of the model.

In the end, B model code is translated into C or Ada code by automatic translator tools. This translation is basically a syntax transformation. However, as translating is here safely critical, usually 2 independent tools are used.

With classical B, the strength of a consistent and fully proved model is so high that it gives a very high level of trust in the code obtained. The proof activity makes it useless to go through a unit test activity, since symbolic proof covers all possible use of each operation, whereas unit tests only cover a limited number of cases.

17.3 B predicate evaluation for data verification and validation

18 Verification with Formal Methods

In the railway domain, the standard EN 50128 highly recommends use of formal methods in requirements specification ([5, Table A.2]), software architecture (A.3), software design and implementation (A.4), verification and testing (A.5), data preparation (A.11), and modelling (A.17) for Safety Integrity Level SIL 3 and above. However, functional/black-box testing is still mandatory in verification; this constraint may be considered as discouraging from the use of formal methods.

Until recently, the situation was quite similar in the aerospace domain. J. Joyce, a member of the RTCA standardisation committee SC-205, described Airbus' problems in certifying their "unit-proof for unit-test" approach:

> "*Formal methods were used for certification credit in development of the A380, but apparently it was not a trivial matter to persuade certification authorities that this was acceptable even with the reference to formal methods in DO-178B as an alternative method.*"

Such experiences eventually caused the more detailed treatment of formal method issues in the revision C of DO-178 that appeared in late 2011. The DO-178C considers formal methods as special cases of reviews and analyses; thus incorporating them without major structural changes of the software development recommendations. For an employed formal method, the standard requires to justify its unambiguity, its soundness[4], and any additional assumptions[5] needed by the method. The DO-178C admits formal property verification on object code as well as on source code, the latter additionally needing evidence about property preservation of the source-to-object code compiler. However, "*functional tests executed in target hardware are always required to ensure that the software in the target computer will satisfy the high-level requirements*" [7, FM.12.3.5].

As a consequence of subsuming formal methods under general reviews and analyses, no deviating special rules to qualify tools are necessary: "*Any tool that supports the formal analysis should be assessed under the tool qualification guidance required by DO-178C and qualified where necessary.*" [7, FM.1.6.2]. Of course, for the railway domain, the rules of EN 50128 for supporting software tools and languages must be taken into account [5, Section 6.7].

During the last 15 years, formal methods have grown out of academic playgrounds and become practically relevant in several applications domains. Below, we sketch a few different tools, also to indicate the variety of issues formal methods can be applied for. Many of the tools mentioned below provide formal verification for programs written in C. There is currently insufficient support for the programming language C++, which is predominantly used in Thales' RBC product. A list of free software tools for formal verification can be found at [38]. The list is not meant to be complete. It is structured by tool purpose, and each tool is briefly introduced.

[4] i.e., that the method never asserts a property to be true when it actually may be not true

[5] e.g. data range limits

18.1 The Frama-C Source Code Analysis Suite

Frama-C [39] is a suite of tools from CEA LIST and INRIA Saclay, dedicated to the analysis of C source code. Frama-C gathers several static analysis techniques in a single collaborative framework. Frama-C also features a formal specification language, ACSL [40], in which the contract of each function of the program can be written (see section 16.2), as well as assertions that are supposed to hold at a given program point.

Frama-C's kernel as well as many analysis plug-ins are available under the LGPL Open-Source licence from [41]. Other plug-ins have been developed by third-party developers, either in an academic [42] or an industrial [23] background. The remainder of this section only deals with the plugins that are released with Frama-C's kernel and are the most relevant for OpenETCS.

Value Analysis

Value analysis is based on abstract interpretation (section 16.1). This plugin analyses a complete application, starting from a given entry point, and gives at each program point an over-approximation of the values that can appear in each memory location at this point. For each operation, Value also checks that whether the abstract value of the operands guarantees that the operation is safe. If this is not the case, it emits an alarm, in the form of an ACSL assertion, and attempts to reduce its abstract state to represent only safe concrete values. If all concrete values are unsafe, then either the corresponding branch of the code is dead (and was only taken because of the over-approximation), or there is a real error in the code. Otherwise, the analysis resumes with the reduced state. Conversely, if no alarm is emitted by Value, the analysed code is guaranteed not to lead to a run-time error.

Value can also be used to check whether ACSL annotations hold or not. However, it is restricted to the subset of the ACSL language that fits well within the abstract representation that is used.

Finally, Value can be tweaked in various ways to increase the precision of the results (leading to fewer false alarms), generally at the expense of the computation time and amount of memory used by the analysis. These options are described in more detail in Value's reference manual [43].

WP

WP is a plugin dedicated to deductive verification (see section 16.2). It uses different models to represent C memory states in the logic. More abstract models lead to easier proof obligations, but cannot be used in presence of low-level pointer arithmetic, while more concrete ones are able to deal with any C construction, at the expense of far more complex proof obligations.

WP has two native interfaces to discharge proof obligations. The first one calls the Alt-Ergo [44] automated theorem prover, while the second let the user do the proof within the Coq [31] interactive proof assistant. In both cases, the original formulas are first run through an internal simplifier, that can directly discharge the simplest proof obligations, without the need for a call to an external tool. In addition, WP can also call the Why3 [30] back-end, through which it has access to a variety of automated provers. Alt-Ergo, Coq and Why3 are available under Open-Source licences (Cecill-C and LGPL). The various possible settings of WP are described in its user manual [45].

While WP's primary usage is to prove functional properties expressed as function contracts, it can also be used to prove the absence of runtime error, either by discharging the alarms emitted

by Value Analysis, or by generating proof obligations for all operations that might lead to a runtime error (without having to use Value first). The latter case is done through the use of the *RTE* plugin, that generates an ACSL assertion for each potentially dangerous operation. WP can then generate proof obligations for these assertions as usual.

Aoraï

While Value and WP are used to verify program properties, the Aoraï plugin is dedicated to generate ACSL specifications (which can then be proved by Value or WP). More precisely, it takes as input an automaton describing the sequence of function calls that are allowed during the execution of a program (from a given entry point). From that automaton, Aoraï instruments the code and provides ACSL contract for each function so that if all the contracts hold, then the code is behaving according to the automaton.

Transitions of the automaton can be guarded by conditions over the state of the program at a given call point. Full syntax of Aoraï's input language is described in [46].

18.2 The Diversity Symbolic Execution Tool

DIVERSITY is a symbolic execution tool developed at $CEA - LIST$. Its underlying technology is *symbolic execution*. Symbolic execution has been first defined for programs [47, 48, 49]. The goal of this technique is to identify, for each possible execution of the program, the constraints to be satisfied in order to follow it. The main idea consists in executing the program, not for concrete numerical values but for symbolic parameters, and to characterize constraints on those parameters at each step of the execution. For instance let us consider that at a given step of an execution the next instruction *ins* to be executed is $if(x > 14)x := x + 1$. Moreover let us suppose that from the previous steps we have computed a couple $(x \rightarrow a, a < 45)$ meaning that before the execution of *ins*, the value of x is represented by the symbolic parameter a, with the constraint that $a < 45$. Executing *ins* results on a new context $(x \rightarrow a + 1, a < 45 \wedge a > 14)$ taking into account both the constraints so that the instruction is executable (x has to be greater than 14 and since x value is a it means that a has to be greater than 14) and the variable updates induced by the instruction (the result of the execution of $x := x + 1$ is that x value is now $a + 1$). $a < 45 \wedge a > 14$ is called a *path condition*. Generating test data to follow some executions comes the to use solvers to find values satisfying such path conditions. Symbolic execution has been later adapted to modeling formalisms like *Input Output Symbolic Transition Systems* ([50, 51]), later to timed version of Input Output Symbolic Transition Systems ([52, 53]) and also to various industrial modeling languages like the sequence diagrams of the UML ([54]). Those symbolic execution adaptations have been used in model based testing contexts. System under test are compared to their models by means of two conformance relations namely *ioco* ([55]) and its timed extension *tioco* ([56]). Those two conformance relations are among the most widely accepted conformance relations. Several testing algorithms were defined based on those conformance relations ([57, 52, 53]).

In the frame of the openETCS project we plan to use DIVERSITY to extract test cases from models defined in the first phases of the system design. Our goal is to extract test cases dedicated to abstract safety requirements. More precisely we focus on safety requirements dealing with communication between sub systems. For that purpose we will use the language of sequence diagrams extended with timing constraints to specify such requirements. With sequence diagrams, one may describe execution scenarios in terms of partially ordered message passing between subsystems. Message passing can be structured thanks to powerful operators expressing sequencing, parallelism, choice, loop... In [54] we show how to automatically analyze such

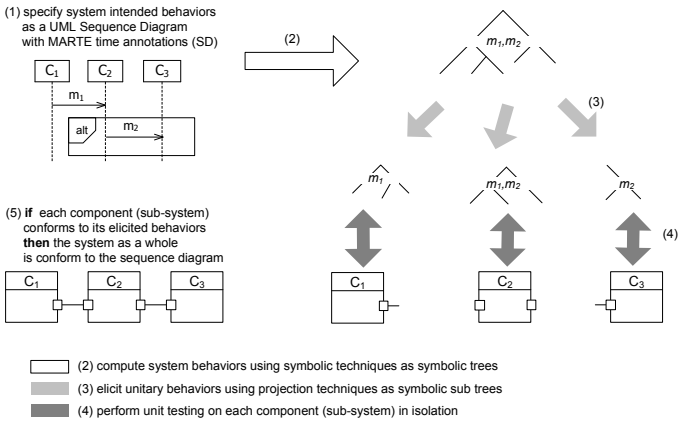


Figure 4. Compositionnal Testing

sequence diagrams with DIVERSITY in order to extract test cases. The originality is that is that, thanks to projection mechanisms, it is possible to extract test cases, not only for the entire system, but also for any of its distinguished sub systems. Thanks to this mechanism, sub systems can be tested as soon as they are implemented, even though the entire system is not yet implemented. In such a process we perform a particular kind of unitary testing in which unit test cases are built according to the usage that will be made of the sub system in the entire system. Faults identified with such an approach are very relevant because we know that they will be activated in the system. The process is illustrated in Figure 4. The first step consists in defining a requirement model in the form of a sequence diagram. The requirement model is analyzed with DIVERSITY in step (2). This analysis results on a so-called *symbolic tree*, whose each path denotes a possible (symbolic) execution of the sequence diagram. Such trees may be theoretically infinite due to the possible occurrences of the "loop" operator of sequence diagrams. Therefore, DIVERSITY uses various stopping criteria to stop the computation (typically based on message coverage notions). The symbolic tree computed in step (2) characterizes executions of the whole system model. However because testing the whole system may be complicated in terms of testing architecture, or simply because one wants to test some sub systems before the whole system is implemented, we offer a mechanisms to extract symbolic trees for each distinguished sub system. This is based on so-called *projection* techniques ([58, 52]). This operation is realized in step (3). In step (4) Each identified sub system is tested thanks to a real time off-line testing algorithm ([53]). Thanks to a compositionality theorem ([59]) we can relate correctness of sub systems and correctness of the whole system (see step 5). The compositionality theorem expresses that, the conformance of each subsystems to all their projections guarantees the conformance of the whole system to the sequence diagram. A direct consequence is that any faults of the whole system can be discovered as a fault of at least one of its sub systems. This implies that testing the whole system mainly comes to test each of its sub systems regardless of a very simple test integration phase in which one only tests that each sub system is correctly connected. We believe that such an approach will be very useful for ETCS systems which are by nature very distributed and thus hardly observable and controllable as a whole. We plan to identify with experts how to partition them into several sub systems that will be more easily observable and controllable at the testing phase.

18.3 Microsoft's Verifier for Concurrent C (VCC)

VCC is a tool from Microsoft Research to prove correctness of annotated concurrent C programs. It was mainly developed to verify Microsoft's *Hyper-V* hypervisor. It supports an own annotation language providing e.g. contracts, pre- and postconditions, and type invariants. It uses the Boogie tool to generate proof obligations, and the automatic prover Z3 to prove them. If an obligation is

violated, the Model Viewer tool can generate a counter-example use case. VCC is available for non-commercial use from [60].

18.4 The Proof Assistants Coq and Isabelle

Coq is an interactive theorem prover and proof checker, developed at INRIA, and based on higher-order logic and the natural deduction calculus. It provides the formal language *Gallina*, in which mathematical definitions can be expressed as well as executable algorithms and theorems. The supporting tool for tactics-based semi-interactive development of proofs is available from [31].

Isabelle, maintained at Cambridge University, and its predecessor *HOL*[6], are similar tactic-oriented interactive theorem provers. Isabelle is available from [32]. While Isabelle is not yet supported in the Frama-C environment, Coq is.

18.5 The Model Checker NuSMV

SMV[7] has been the first model checker based on binary decision diagrams. *NuSMV* is a reimplementation by the Fondazione Bruno Kessler that is in addition capable of performing SAT-based model-checking. It supports both Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). NuSMV's source code is available under an LGPL license from [61].

18.6 Formal Verification of Real-Time Aspects based on Timed Automata

Verifying system properties involving time is difficult with traditional model checking methods. Commonly used *temporal logics*, such as LTL or CTL catch discrete and qualitative aspects of time and allow to formulate properties such as[8]

X At the next point in time a property holds.

F At some future point in time a property holds.

G Always/generally (now and at any future point in time) a property holds.

U A property p holds until a property q holds.

While it is possible to state properties that must be satisfied at individual (discrete) points in time, continuous and quantitative aspects of time as in the safety requirement

> "The delay between receiving an emergency message and the issuing of a brake order is less than 1 second."

are a real challenge. The problem does not stem from discrete vs. continuous time, as any physical realisation of a real-time system is inherently discretised by its clock. Instead, an operator for expressing arbitrary temporal quantities or differences is missing. Thus, for LTL and a clock of 1 kHz, it would be required to use the operator **X** 1000 times. This notation is rather unhandy, as it enforces to express a functional property relative to a particular system.

[6] Higher Order Logic
[7] Symbolic Model Verifier
[8] The prefixes are the corresponding linear time operators.

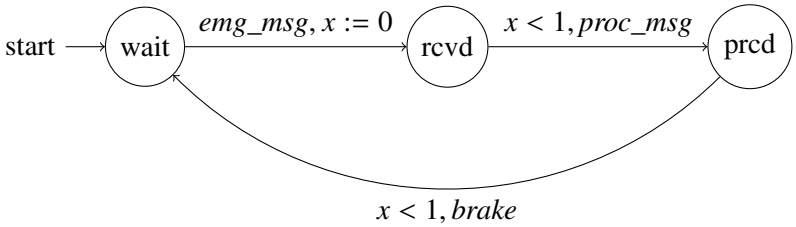


Figure 5. First (faulty) version of a timed automaton for processing emergency messages

This lack of expressivity is not merely a matter of notation, i.e., LTL or CTL, but also of the underlying semantics. Before introducing a better suited logic we will first consider a formalism that serves as this logic's semantics – namely *timed automata*.

Timed Automata

Timed automata [62] are essentially finite automata extended with a finite set of clocks that all proceed at the same rate. Clocks may be individually reset to zero. Clock variables can be part of constraint expressions that may be used as transition guards. A transition can only be taken if its guard is fulfilled. Similarly, it is possible to specify an invariant for a state that must be satisfied when the automaton is in this state. Thus, we can enforce time constraints for the runs of the automaton.

An Example

The timed automaton in Figure 5 depicts an automaton representing an over-simplified version of an OBU subsystem processing emergency messages. It has three states, one clock x and three actions, *emg_msg* (reception of an emergency message), *proc_msg* (processing the message, e.g. raising an alarm) and *brake* (issueing the brake order). Upon receiving an emergency message the clock x is reset. The *proc_msg*-transition is guarded by the clock constraint $x < 1$ preventing the transition to be taken if $x \geq 1$. The same holds for the *brake*-transition.

One might think that the automaton from Figure 5 thus fulfills the safety requirement stated above. Due to the operational semantics of timed automata this is not true: a timed automaton in a given state can either take a transition or wait for an arbitrary amount of time. Thus, if automaton waits in state rcvd and x exceeds one second, the system will deadlock as the next transition is guarded by the constraint $x < 1$. A run of the automaton that could serve as counterexample is, e.g.

$$(\text{wait}, 0) \rightarrow (\text{wait}, 0.5) \rightarrow (\text{rcvd}, 0.5) \rightarrow (\text{rcvd}, 2) \rightarrow \text{DEADLOCK}$$

A solution to this problem is to force the automaton to proceed by placing *progress* constraints on the states. This has been done in Figure 6. The transition guards have been omitted as they are not necessary anymore. Now the safety property "The delay between receiving an emergency message and the issuing of a brake order is less than 1 second." is fulfilled.

UPPAAL

Uppaal is a tool for modelling and verifying timed automata developed by the universities of Uppsala and Aalborg [63]. This toolkit is under constant development and comes with an

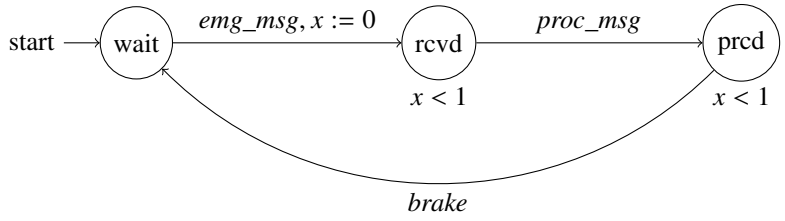


Figure 6. Corrected version of the timed automaton for processing emergency messages

academic as well as a commercial licence. Moreover, there is a comparably large body of literature featuring UPPAAL, providing introductory and industrial examples.

UPPAAL extends timed automata with synchronisation enabling concurrent, communicating automata representing different parts of a system. In addition, variables other than clocks are supported making the modelling language more powerful. The logic used for expressing real-time properties is a subset of TCTL (Timed Computation Tree Logic). Nesting of temporal operators is not supported leading to a restriction in expressiveness.

From SysML/UML to Timed Automata

Timed automata and statecharts in SysML/UML are both based on the concept of finite automata. Thus, it seems reasonable to extract timed automata from existing statecharts which is addressed in the literature [64, 65, 66]. In this way, safety properties – formalised as TCTL formulae – can be verified in an automated fashion for a given statechart. However, there remain challenges:

- Translating hierarchical states to timed automata is not straight-forward and complicates matters significantly. If an hierarchical state-chart is flattened, structural information is lost and makes the timed automaton more difficult to read and understand. Thus, it is advisable to retain some kind of hierarchy, possibly by using synchronisation mechanisms.
- Special statechart features, such as history nodes that have a partially undefined semantics according to the current SysML/UML standard [67], introduce problems. As they are not used very often, they can possibly left out in a first iteration.

19 Verification with Model-Based Simulation

This section addresses verification based on simulation. By building an executable model of system components its real-time behaviour can by analysed and evaluated before actually building the entire system.

19.1 Modelling with SysML and SystemC

Specifications in natural language are difficult to handle. Breaking down an overall system description into small comprehensible parts reduces complexity and eases interdisciplinary communication to be more efficient in performing development tasks.

SysML, developed by the OMG (Object Management Group), is a simple but powerful general-purpose graphical modeling language that does not directly support executable models. However, there is a variety of tools for code generation from UML/SysML, especially for the Eclipse platform and the Papyrus framework that will be used in the project.

To enable model execution and especially real-time simulation it is considered to generate semi-formal SystemC code from that abstract SysML models. It has to be investigated whether the the SysML model have to be adapted to a domain or language specific version. Concrete analysis will show whether this is feasible.

SystemC is a C++ library providing an event-driven simulation interface suitable for electronic system design at various abstraction levels (from high level down to individual hardware components). It enables a system designer to simulate concurrent processes. SystemC processes can communicate in a simulated real-time environment, using channels of different datatypes (all C++ types and user defined types are supported). SystemC supports hardware and software synthesis (with the corresponding tools). SystemC models are executable.

19.2 Model execution and simulation

The aim of the execution of an SystemC model is to ensure that the working capacity (performance) of the underlying hardware system is sufficient to meet the system requirements. It has to be analysed which hardware resources will be needed for the OBU to avoid excessive delays and to ensure adequate response times in critical situations. Because of the integrated simulation enviroment, SystemC enables scheduling analysis for average and worst-case conditions and provides analyses of process resources for individual system functions.

In addition, by creating an executable system model from SysML or UML, the (real time) behaviour of the system can be analysed which is not feasible at the SysML level.

References

[1] I. de la Torre. Project quality assurance plan. openETCS Deliverables D1.3.1.

[2] M. Behrens, H. Hungar, A. Cavalli, J. Gerlach, H. Manz, and C. Cornu. openETCS validation and verification strategy work package: Description of work, May 2013.

[3] Jens Gerlach, Virgile Prevosto, Jochen Burghardt, Kerstin Hartig, Kim Völlinger, and Hans Pohl. Formal Specification and Automated Verification of Railway Software with Frama-C. In *IEEE International Conference on Industrial Informatics (INDIN)*. IEEE Xplore, July 2013.

[4] CPN Tools. `http://cpn-tools.org`.

[5] CENELEC, European Committee for Electrotechnical Standardization. EN 50128: Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems, June 2011.

[6] RTCA SC-205. *Software Considerations in Airborne Systems and Equipment Certification (DO-178C)*. Radio Technical Commission for Aeronautics (RTCA Inc.), Washington/DC, Dec 2011.

[7] RTCA SC-205. *Formal Methods Supplement to DO-178C and DO-278A (DO-333)*. Radio Technical Commission for Aeronautics (RTCA Inc.), Washington/DC, Dec 2011.

[8] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 38(2-3):258–287, 1999. Reprint from 1976.

[9] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Softw.*, 13(6):47–55, 1996.

[10] Carnegie Mellon University Software Engineering Institute. Architecture tradeoff analysis method. http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm.

[11] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with smt-solving and abstract interpretation. In Bobaru et al. [68], pages 298–312.

[12] Jan Peleska, Artur Honisch, Florian Lapschies, Helge Löding, Hermann Schmid, Peer Smuda, Elena Vorobev, and Cornelia Zahlten. A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In Burkhart Wolff and Fatiha Zaidi, editors, *Testing Software and Systems. Proceedings of the 23rd IFIP WG 6.1 International Conference, ICTSS 2011*, volume 7019 of *LNCS*, pages 146–161, Heidelberg Dordrecht London New York, November 2011. IFIP WG 6.1, Springer.

[13] Helge Löding and Jan Peleska. Timed moore automata: test data generation and model checking. In *Proc. 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*. IEEE Computer Society, 2010.

[14] A. Pnueli, O. Shtrichman, and M. Siegel. The code validation tool CVT: Automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[15] J. Peleska, J. Feuser, and A. E. Haxthausen. *Railway Safety, Reliability and Security: Technologies and Systems Engineering*, chapter The Model-Driven openETCS Paradigm for Secure, Safe and Certifiable Train Control Systems, pages 22–52. In Flammini [69], 2012.

[16] Anne Elisabeth Haxthausen, Jan Peleska, and Sebastian Kinder. A formal approach for the construction and verification of railway control systems. *Formal Asp. Comput.*, 23(2):191–219, 2011.

[17] Verified Systems International GmbH. Verified :: Products. http://www.verified.de/en/products.

[18] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with smt-solving and abstract interpretation. In Mihaela Bobaru, Klaus Havelund, GerardJ. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 298–312. Springer Berlin Heidelberg, 2011.

[19] Jörg Brauer, Jan Peleska, and Uwe Schulze. Efficient and trustworthy tool qualification for model-based testing tools. In Brian Nielsen and Carsten Weise, editors, *Testing Software and Systems*, volume 7641 of *Lecture Notes in Computer Science*, pages 8–23. Springer Berlin Heidelberg, 2012.

[20] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd Int. Symp. on Programming*, pages 106–130, Paris, 1976. Dunot.

[21] Jean Souyris and David Delmas. Experimental Assessment of Astrée on Safety-Critical Avionics Software. In *Proc. Int. Conf. Computer Safety, Reliability, and Security, SAFECOMP 2007*, volume 4680 of *LNCS*. Springer, September 2007.

[22] Bernhard Beckert and Claude Marché, editors. *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *Lecture Notes in Computer Science*. Springer, 2010.

[23] Dillon Pariente and Emmanuel Ledinot. *Formal Verification of Industrial C Code using Frama-C: A Case Study*, pages 205–219. Volume 6528 of Beckert and Marché [22], 2010.

[24] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.

[25] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[26] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, October 1969.

[27] C.A.R. Hoare and Niklaus Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2:335 – 355, 1973.

[28] Atelier B. `http://www.atelierb.eu/`.

[29] Event-B. `http://www.event-b.org/`.

[30] Prover Platform. `http://why3.lri.fr`.

[31] Coq Prover. `http://coq.inria.fr`.

[32] Isabelle Theorem Prover. `http://www.cl.cam.ac.uk/research/hvg/isabelle`.

[33] Edmund M. Clarke and Bernd-Holger Schlingloff. Model checking. In Robinson and Voronkov [34], pages 1637–1790.

[34] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier, 2001.

[35] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS'99, pages 193—207. Springer-Verlag, 1999.

[36] A Bradley. SAT-based model checking without unrolling. *\ldots , Model Checking, and Abstract Interpretation*, 2011.

[37] K.L. McMillan. Interpolation and sat-based model checking. In Jr. Hunt, WarrenA. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 2003.

[38] List of free formal verification tools. `http://gulliver.eu.org/free_software_for_formal_verification`.

[39] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: a Software Analysis Perspective. In *Proceedings of Software Engineering and Formal Methods (SEFM)*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.

[40] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*, 1.7 edition, April 2013. available at `http://frama-c.com/download/acsl.pdf`.

[41] Frama-C: Source Code Analysis Suite. `http://frama-c.com/`.

[42] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *Proceedings of Programming Languages Design and Implementation (PLDI)*, 2011.

[43] Pascal Cuoq, Boris Yakobowski, and Virgile Prevosto. *Frama-C's value analysis plug-in*. CEA LIST, fluorine-20130601 edition, June 2013. available at `http://frama-c.com/download/frama-c-value-analysis.pdf`.

[44] Theorem Prover. `http://alt-ergo.lri.fr`.

[45] Patrick Baudin, Loïc Correnson, and Zaynah Dargaye. *WP Plugin*. CEA LIST, 0.7 for fluorine-20130601 edition, 2013. available at `http://frama-c.com/download/frama-c-wp-manual.pdf`.

[46] Nicolas Stouls and Virgile Prevosto. *Aoraï Plugin Tutorial*. INSA Lyon and CEA LIST, 2013. Available at `http://frama-c.com/download/frama-c-aorai-manual.pdf`.

[47] J.-C. King. A new approach to program testing. *Proceedings of the international conference on Reliable software, Los Angeles, California*, 21-23:228–233, April 1975.

[48] L.-A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, 2(3):215–222, September 1976.

[49] C.-V. Ramamoorthy, S.-F. Ho, and W.-T. Chen. On the automated generation of program test data. *IEEE Transactions on software engineering*, 2(4):293–300, September 1976.

[50] N. Rapin, C. Gaston, A. Lapitre, and J.-P. Gallois. Behavioral Unfolding of Formal Specifications Based on Communicating automata. In *Proceedings of the 1st workshop on Automated Technology for Verification and Analysis (ATVA)*, 2003.

[51] C. Gaston, M. Aiguier, and P. Le Gall. Algebraic Treatment of Feature-oriented Systems. In *Language Constructs for Describing Features*. Springer LNCS, 2000.

[52] J.P. Escobedo, C. Gaston, and P. Le Gall. Timed Conformance Testing for Orchestrated Service Discovery. In *Proceedings of the 8th International Symposium on Formal Aspects of Component Software (FACS)*. Springer LNCS, 2011.

[53] B. Bannour, J.P. Escobedo, C. Gaston, and P. Le Gall. Off-line test case generation for timed symbolic model-based conformance testing. In *Proceedings of the 23rd International Conference on Testing Software and Systems (ICTSS)*. Springer LNCS, 2012.

[54] B. Bannour, C. Gaston, and D. Servat. Eliciting unitary constraints from timed Sequence Diagram with symbolic techniques: application to testing. In *Proceedings of the 18th Asian-Pacific Software Engineering Conference (APSEC)*. IEEE Computer Society, 2011.

[55] J. Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[56] M. Krichen and S. Tripakis. Black-box time systems. In *Proc. of Int. SPIN Workshop Model Checking of Software*. Springer, 2004.

[57] C. Gaston, P. Le Gall, N. Rapin, and A. Touil. Symbolic execution techniques for test purpose definition. In *Proceedings of the 18th International Conference on Testing Communicating Systems (TestCom)*. Springer LNCS, 2006.

[58] A. Faivre, C. Gaston, and P. Le Gall. Symbolic Model Based Testing for Component Oriented Systems. In *Proceedings of the 19th International Conference on Testing Communicating Systems (TestCom/FATES)*. Springer LNCS, 2007.

[59] B. Bannour. *Symbolic analysis of scenario based timed models for component based systems: Compositionality results for testing*. PhD thesis, CEA LIST / École Centrale Paris, 2012. `http://www.cti.ecp.fr/~bannourb/PhDthesis.pdf`.

[60] Microsoft's Verifier for Concurrent C. `http://research.microsoft.com/en-us/projects/vcc`.

[61] NuSMV Model Checker. `http://nusmv.fbk.eu`.

[62] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.

[63] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[64] Alexandre David, M.Oliver Möller, and Wang Yi. Formal verification of uml statecharts with real-time extensions. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2002.

[65] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking - timed uml state machines and collaborations. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2*, FTRTFT '02, pages 395–416, London, UK, UK, 2002. Springer.

[66] Karsten Diethers and Michaela Huhn. Vooduu: Verification of object-oriented designs using uppaal. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 139–143. Springer, 2004.

[67] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem-Paul de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In Kung-Kiu Lau and Richard Banach, editors, *Formal Methods and Software Engineering*, number 3785 in Lecture Notes in Computer Science, pages 52–65. Springer Berlin Heidelberg, January 2005.

[68] Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors. *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*. Springer, 2011.

[69] Francesco Flammini, editor. *Railway Safety, Reliability and Security: Technologies and Systems Engineering*. Information Science Reference, 2012.

Appendix A: Requirements on Verification & Validation

A.1 Requirements on Verification & Validation from D2.9

The already provided requirements require a safety plan compliant to the CENELEC EN 50126, 50128 and 50129. This pulls a number of requirements on V&V, including Verification and Validation plans. On the topic of compliance to EN 50128, one shall also refer to the D2.2 document.

R-WP2/D2.6-02-061 A Verification plan shall be issued and complied with.

R-WP2/D2.6-02-061.01 The verification plan shall provide a method to demonstrate the requirements covering all the development artifacts.

R-WP2/D2.6-02-061.02 The verification plan shall state all verification activities required for each of these development artifacts.

R-WP2/D2.6-02-062 A Validation Plan shall be issued and complied with.

R-WP2/D2.6-02-062.01 The validation plan shall provide a method to validate all functional and safety requirements over all development artifacts.

R-WP2/D2.6-02-062.02 The validation plan shall state all validation activities required for each of these development artifacts.

R-WP2/D2.6-01-021 The test plan shall comply the mandatory documents of the SUBSET-076, restricted to the scope of the OpenETCS project.

Justification. It will possibly be difficult to model all the tests in the course of the project, but the test plan should at least be complete.

R-WP2/D2.6-02-063 Each design artifact needs a reference artifact which it implements (*e.g.* code to detailed model, SFM to SSRS model. . .)

R-WP2/D2.6-02-063.01 The implementation between them relation shall be specified in detail.

e.g. for state machine and a higher level state machine mapping of interfaces, states and transition is required. This includes additional invariants, input assumptions and further restrictions. This informaiton is the basis for verification activities.

R-WP2/D2.6-02-063.02 The design of the artifacts shall be made such to allow verifiability as far as possible.

R-WP2/D2.6-02-064 The findings from the verification shall be traced, and will be adequately addressed (taken into consideration, or postponed or discarded with a justification).

A.2 General Requirements on Verification

Excerpt from EN 50128:2011 [N01]	Requirement	Project Relevance
5.3.2.7	For each document, traceability shall be provided in terms of a unique reference number and a defined and documented relationship with other documents.	fully applicable
5.3.2.8	Each term, acronym or abbreviation shall have the same meaning in every document. If, for historical reasons, this is not possible, the different meanings shall be listed and the references given.	
5.3.2.9	Except for documents relating to pre-existing software (see 7.3.4.7), each document shall be written according to the following rules: • it shall contain or implement all applicable conditions and requirements of the preceding document with which it has a hierarchical relationship; • it shall not contradict the preceding document.	
5.3.2.10	Each item or concept shall be referred to by the same name or description in every document.	
6.5.4.14	Traceability to requirements shall be an important consideration in the validation of a safety-related system and means shall be provided to allow this to be demonstrated throughout all phases of the lifecycle.	
6.5.4.15	Within the context of this European Standard, and to a degree appropriate to the specified software safety integrity level, traceability shall particularly address a) traceability of requirements to the design or other objects which fulfil them, b) traceability of design objects to the implementation objects which instantiate them. c) traceability of requirements and design objects to the tests (component, integration, overall test) and analyses that verify them. Traceability shall be the subject of configuration management.	

Excerpt from EN 50128:2011 [N01]	Requirement	Project Relevance
6.5.4.16	In special cases, e.g. pre-existing software or prototyped software, traceability may be established after the implementation and/or documentation of the code, but prior to verification/validation. In these cases, it shall be shown that verification/validation is as effective as it would have been with traceability over all phases.	This requirement does not apply to the project.
6.5.4.17	Objects of requirements, design or implementation that cannot be adequately traced shall be demonstrated to have no bearing upon the safety or integrity of the system.	

Excerpt from EN 50128:2011 [N01]	Requirement
6.1.4.1	Tests performed by other parties such as the Requirements Manager, Designer or Implementer, if fully documented and complying with the following requirements, may be accepted by the Verifier.
6.1.4.2	Measurement equipment used for testing shall be calibrated appropriately. Any tools, hardware or software, used for testing shall be shown to be suitable for the purpose.
6.1.4.3	Software testing shall be documented by a Test Specification and a Test Report, as defined in the following.
6.2.4.2	A Software Verification Plan shall be written, under the responsibility of the Verifier, on the basis of the necessary documentation.
6.2.4.3	The Software Verification Plan shall describe the activities to be performed to ensure proper verification and that particular design or other verification needs are suitably provided for
6.2.4.4	During development (and depending upon the size of the system) the plan may be subdivided into a number of child documents and be added to, as the detailed needs of verification become clearer.
6.2.4.5	The Software Verification Plan shall document all the criteria, techniques and tools to be used in the verification process. The Software Verification Plan shall include techniques and measures chosen from Table A.5, Table A.6, Table A.7 and Table A.8. The selected combination shall be justified as a set satisfying 4.8, 4.9 and 4.10
6.2.4.6	The Software Verification Plan shall describe the activities to be performed to ensure correctness and consistency with respect to the input to that phase. These include reviewing, testing and integration.
6.2.4.7	In each development phase it shall be shown that the functional, performance and safety requirements are met.
6.2.4.8	The results of each verification shall be retained in a format defined or referenced in the Software Verification Plan.

Excerpt from EN 50128:2011 [N01]	Requirement
6.2.4.9	The Software Verification Plan shall address the following: a) the selection of verification strategies and techniques (to avoid undue complexity in the assessment of the verification and testing, preference shall be given to the selection of techniques which are in themselves readily analysable); b) selection of techniques from Table A.5, Table A.6, Table A.7 and Table A.8; c) the selection and documentation of verification activities; d) the evaluation of verification results gained; e) the evaluation of the safety and robustness requirements; f) the roles and responsibilities of the personnel involved in the verification process; g) the degree of the functional based test coverage required to be achieved; h) the structure and content of each verification step, especially for the Software Requirement Verification (7.2.4.22), Software Architecture and Design Verification (7.3.4.41, 7.3.4.42), Software Components Verification (7.4.4.13), Software Source Code Verification (7.5.4.10) and Integration Verification (7.6.4.13) in a way that facilitates review against the Software Verification Plan.

A.3 Glossary

API: Application Programming Interface. In the project, the API defines the interface of the EVC software to the operating system and hardware. *The exact nature of the API still needs to be defined, whether it should be seen as a specification or as an implementation has yet to be resolved.*

ATAM: Architecture Tradeoff Analysis Method

DAS2V: Design Artifact Subject to Verification or Validation, e.g. some model or code fragment which has to be verified against its specification.

EVC: European Vital Computer

FLOSS: Free/Libre/Open Source Software

FFM: Fully Formal Model. Sometimes called “Strictly Formal Model”. A model of a part of the design which has a fully formal semantics and can thus be subjeced to rigorous analysis methods from the domain of mathematical or computational logic.

HW: Hardware

SAAM: Software Architecture Analysis Method

SFM: Semi Formal Model. A model of some part of the design whose semantical interpretation is either not fully fixed or is similar to that of a program. I.e., the interpretation might depend on variations in the code generation or compilation, or it does not resolve “semantic variation points” (UML).

SW: Software