OETCS/WP4/D4.3.1

openETCS

ITEA2 Project
Call 6 11025
2012 – 2015

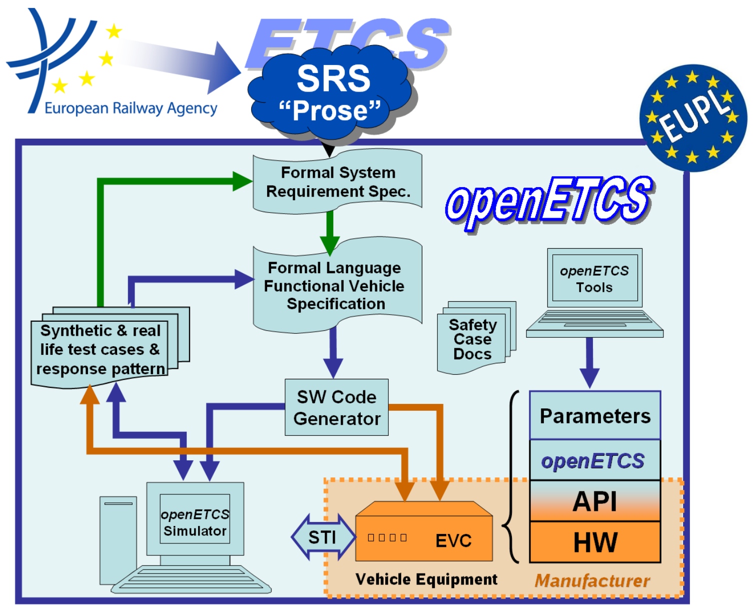
openETCS@ITEA Work Package 4.2: “Verification & Validation of the Formal Model”

# D4.3.1 1st interim V&V report on the applicability of the V&V approach to the formal abstract model

**Version 04**

Marc Behrens, Ana Cavalli, João Santos, Huu-Nghia Nguyen, Stefan Rieger, Cécile Braunstein, Uwe Steinke, Benoît Lucet, Matthias Güdemann, Brice Gombault, Marielle Petit-Doche, Alexander Nitsch & Benjamin Beichler, Silvano Dal Zilio, Ning Ge and Marc Pantel

January 2014
revised November 2015



**Funded by:**

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

This page is intentionally left blank

**openETCS@ITEA Work Package 4.2: "Verification & Validation of the Formal Model"**

**OETCS/WP4/D4.3.1**
**January 2014**
**revised November 2015**

# D4.3.1 1st interim V&V report on the applicability of the V&V approach to the formal abstract model

**Version 04**

Document approbation

| Lead author: | Technical assessor: | Quality assessor: | Project lead: |
|---|---|---|---|
| location / date | location / date | location / date | location / date |
| signature<br>() | signature<br>() | signature<br>() | signature<br>Klaus-Rüdiger Hase<br>(DB Netz) |

Marc Behrens

Deutsches Zentrum für Luft und Raumfahrt e.V.

Ana Cavalli, João Santos and Huu-Nghia Nguyen

Institut Mines-Télécom

Stefan Rieger

TWT GmbH Science & Inovation

Cécile Braunstein

Uni Bremen

Uwe Steinke

Siemens

Benoît Lucet, Matthias Güdemann, Brice Gombault and Marielle Petit-Doche

Systerel

Alexander Nitsch & Benjamin Beichler

University of Rostock

Silvano Dal Zilio and Ning Ge

LAAS-CNRS

Marc Pantel

INPT

Prepared for openETCS@ITEA2 Project

**Abstract:** This document presents the final of verification and validation of the formal model. Specifically, the following sections present the contributions of partners of WP 4:

- Section **??** — Institut Mines-Télécom: Verification of the Movement Authority (Subset-026 chapter 3.8)
- Section **??** – TWT GmbH Science & Innovation: Verification of Procedures (Subset-026 chapter 5)
- Section **??** – University of Bremen: Verification of the Management of the Radio Comumunication (Subset-026 chapter 3.5)
- Section **??** – University of Rostock: Verification of the Speed and distance Monitoring (Subset-026 chapter 3.13)
- Section 2 – Systerel: Verification of Procedure on-Sight (Subset-026 chapter 5) and the Management of Radio Communication (Subset-026 chapter 3.5)
- Section 3 –LAAS and INPT: Verification of the Speed and Distance Monitoring (Subset-026 chapter 3.13)

# Table of Contents

# Figures and Tables

# 1 Introduction

This verification evaluated in five different approaches that the design conforms to an excerpt of Subset-026 [1] related to the specification [2] [3]. It is the first of three reports. It is performed before a design or development process is available. The design used for verification in this report is created and adapted to improve later results of verification. Due to the applicable restrictions the results of the first report focus on the verification itself rather than to give a verdict about conformity of design. All following reports will focus solely on the design provided by the work package "Modelling – Code Generation".

To ensure the correctness and consistency of a design model and its implementation, the validation and verification has to be performed alongside with the modeling process. As the model and code of the EVC are produced by WP3, thus this task will be performed repeatedly during WP3 and will provide feedback to it.

# 2 Systerel

Three approaches of VnV on formal models have been experimented and are presented in this section:

1. VnV on classical B model that cover software design level, in the objective to provide an open-source approach;
2. VnV on Event-B model that cover system level and support safety analysis;
3. VnV on Scade model that cover software design level.

Classical B and Scade model specify the same example of Subset-026 chapter 5.9 “Procedure on-Sight”, Event-B model specifies the “Mangement of Radio communication” function.

## 2.1 Verification and Validation on Classical B model

The first section 2.1.1 describes usual verification and validation activities applied during the design of industrial critical software using the B method. Second section 2.1.2 gives a description of tools available.

Last section 2.1.3 describes the results on an example.

### 2.1.1 Verification processes applicable to a B model

A B model is a textual and formal specification covering the functional behaviour of a safety critical software. It is usually written based on a high-level specification (informal or formal specification, for example SysML or a natural language). It is gradually refined, starting at the top with an abstract notation and ending at the bottom with sequential instructions — which are then automatically translated in a target language such as C or Ada.
Thus, we define three objects of verification and validation: the specification, the B model and the generated source code.

Validation consists of:

- guaranteeing the functional adequacy between the specification and the model (this can be achieved, for example, through review and proofreading),
- building a test environment around the generated source code and test it.

Hence, this section will mainly focus on verification, i.e. the methods and tools required to assure that B method is a consistent way of producing critical software.

In this section, we demonstrate the suitability of the B method towards the problematics encountered in the openETCS project by introducing the different verification processes applicable to a B model.
Each of the verification process is presented and its contributions to the system security and consistency are discussed.

**Type checking** Static type checking (TC) is a basic form of program verification that ensures type safety of the model. It is the first verification — after lexical and syntactic analysis — to be performed on a B model, thus allowing an early detection of problems. It is also a pre-requisite for the higher-level verifications.
Strong typing ensures a consistent use of data and is essential to writing correct formulae (predicates or expressions).

The type checking process consists of two main activities: data typing and type verification.
*Data typing* is the activity of assigning a type to newly-encountered data in a predicate or a substitution. In B, a substitution is comparable to a set of instructions that modify variables. For more information, see [4]. It is based on an inference mechanism, which is able to deduce the type of a newly-encountered variable from the type of the other variables intervening in the predicate or the substitution, and specific inference rules.
On the other hand, *type verification* is the activity of verifying typing rules between already-typed variables. These rules are specific to their applying predicates, expressions or substitutions.

**B0-check** The B0 verification has the specific purpose of checking the respect of the rules that the B model has to conform to in order to generate the translation to C or ADA. These rules are called implementability rules and must be respected in order for the translation to process properly. They also ensure that the resulting code is executable and respects a set of properties.

**Well-definedness** An expression is well-defined (WD) if its associated value or interpretation exists and is unique, thus avoiding ambiguity.
Examples of ill-defined formulae include division by zero, function application to an argument out of the domain, function application of a relation etc.

Well-definedness checking is thus an extra verification that helps strengthen the model.

**Model checking** Model checking is a static semantic check that searches for invariant or assertion violations and deadlock states. It exhaustively explores the whole state space, i.e., is in general limited to finite systems.
This type of verification animates the model, modifying the current state of the machine, starting from the initialization. Operation calls are simulated and modify the internal state which is then checked for various properties. Most of the time, an invariant or assertion violation is looked for.
This verification process, as opposed to the ones previously introduced, considers the semantics of the model and aims at verifying properties dynamically. However, it has its limitations:

- inability to run through all of the states and transitions for models with infinitely many states
- difficulty to deal with very large models or large domains (e.g. 32 bit integers) often leading to state-space explosion because of exponential growth of the state space size.

This means that potential erroneous states can be missed, and that this verification process is not sufficient to ensure *correctness*, though satisfying as an additional verification tool.

**Constraint-based checking** Constraint-based checking (CBC) is the process of finding a given valid state, for which an operation call leads to an erroneous state. This is done by constraint solving instead of — as seen for model checking — running through states from the initialization. This technique will usually provide more counter-examples than model-checking, because it ignores the initialization constraints and can thus reach a wider range of states.

**Formal proof** A proof obligation is a mathematical lemma generated by the proof obligation generator (POG). It corresponds to a consistency property of the model, that has to be demonstrated. These properties are either generated to ensure correctness of the model, e.g., refinement, variable typing, well-definedness or they are specified manually as invariants of the system, e.g., dependent typing, safety invariants. A fully-proved model is said to be *correct*, in the sense that every property (invariant, assertion) expressed is proved to hold for every state of the program. If a proof obligation is not provable, it means that the B model is inconsistent and must be corrected. In fact, the goal of any B development is to obtain a proved model.
In contrast to model-checking, formal proof does not require to make assumptions about the size of the system (number of transitions). It is reliable and powerful, but it needs to be taken into account that:

- some proofs can be very difficult to solve,
- the model needs to be written as to make it the simplest to prove, which demands experience and skill.

A proved model will always meet the formalized safety and security qualifications ; however that does not mean it will behave in regards to the informal specification! It must be validated that the formalized specification (and in consequence the formalized model) correctly implements the informal specification, in particular that the intended functioning is possible. This is the domain of validation, as discussed in the introduction.

### 2.1.2 Tools

In this section, we present the existing tools suitable for the verification processes defined in Section 2.1.1.

**Atelier B** Atelier B[1] is the main development tool associated with the B method and is produced and distributed by ClearSy. It provides most of the needed tools.

**B compiler**
The B compiler performs syntax analysis, type checking, identifier scope resolution and B0-checking. It is part of Atelier B as an open source tool.

**Proof obligation generator**
Atelier B's POG is currently the only known fully operational POG for B, and is free of charge — although proprietary software, which means closed-source. ClearSy is currently working on a new proof obligation generator ; whether it will be open source or not is to be determined.

[1]See `http://www.atelierb.eu/en`

**Prover, proof assistant, user-defined rules**
Atelier B provides a free of charge — although not open source — prover which discharges proof obligations. Depending on the complexity of the model, a varying proportion of the proof obligations is discharged automatically.
For the remaining proof obligations, Atelier B provides an interactive proof assistant allowing the user to guide the prover in discharging the PO. The user may define theories (or rules) which have in turn to be proved. The user-defined rules are organized in a database and can be automatically verified and validated with the Rule Verifier of Atelier B.

**Atelier B translators**
Translators are an essential component of the industrial success of B. The translators take the B0 implementations as input and produce a target source code, typically Ada or C/C++, ready to be compiled or integrated in an environment.
ClearSy provides an open source translator, but it does not reach the T3 level of qualification[2].

**ProB** ProB is an animator and model checker for B models distributed under the EPL license (open source) and mainly developed by Formal Mind[3].
It performs model checking as well as constraint-based checking and searches for a range of errors, with customizable search options and various graphical views. ProB also handles automatic coverage reports generation.
ProB is a mature tool and is being used by several industrials such as Siemens and Alstom. This makes it a precious tool for the verification processes described above.

**Tool qualification** Atelier B has been used for many years to develop railway critical software. It is, for this exact reason, *qualified* by the main actors of the railway domain: SNCF, RATP, Alstom, Siemens etc.

The CENELEC norm defines qualification levels for verification tools. Annex A 5 of the norm specifies several verification techniques and for each of them, a recommendation level (mandatory, highly recommended, recommended). Below are listed the different techniques and measures along with their recommendation levels for SIL4 developments : Recommended, Highly Recommended or Mandatory.

Table 1 shows, for each of the verification processes presented in 2.1.1 (specification and source code were added as artifacts which support the activity), the corresponding item in the CENELEC norm annex table.

A B model proof provides a formal proof in the form of a proof tree that can be inspected by humans and is machine-checkable, i.e., an automated program can replay the proof steps and verify the correct application of the proof rules.
Static type checking, B0 compliance for programming language translation and constraint based checking do not require any dynamic, i.e., state-space information to detect problems. Therefore these representing static analysis approaches of B model.
Model checking analyzes the dynamic behavior of the model by generating and exploring the state-space. Very often, model checkers can generate counter-examples for violated properties

[2]For additional information on qualification, see subsection 2.1.2 or the CENELEC norm.
[3]See `http://www.formalmind.com`

| | | level | spec | TC | B0C | MC | CBC | proof | source code |
|---|---|---|---|---|---|---|---|---|---|
| A 5.1 | Formal Proof | HR | | | | | | ✓ | |
| A 5.2 | Static Analysis | HR | | ✓ | ✓ | | ✓ | | |
| A 5.3 | Dynamic Analysis and Testing | HR | | | | ✓ | | | |
| A 5.4 | Metrics | R | | | | | | | |
| A 5.5 | Traceability | M | ✓ | | | | | ✓ | |
| A 5.6 | Software Error Effect Analysis | HR | | | | | | | |
| A 5.7 | Test Coverage for code | HR | | | | | | | |
| A 5.8 | Functional/ Black-box Testing | M | | | | | | | ✓ |
| A 5.9 | Performance testing | HR | | | | | | | |
| A 5.10 | Interface testing | HR | | | | | | | |

**Table 1. Correspondence between CENELEC norm recommendations and the presented verification processes**

and can therefore be applied as a testing method.
The specification in the B model represents a formalized version of the specification, the proof trees use formalized properties as proof steps. These can be traced in the informal specification. Finally the translated source code can be compiled and tested using various testing methods, including functional tests and black-box testing.

**Conclusion on tools** Table 2 summarizes the presentation of the tools in subsections 2.1.2 and 2.1.2.
Atelier B and ProB are both mature tools that have proved their worth. They are the core tools for validation processes of B models. However, key components of Atelier B are not open source and this issue is not completely compensated by ProB's model checking and CBC.
An ongoing research project named BWare[4] and conducted by ClearSy, Inria, LRI and others aims at providing a framework from proof obligation generation to proof discharge by the means of SMT solvers. This promising project started in September 2012 and is funded for a period of four years. It opens perspectives for the near future in terms of open source B model verification.

| | TC | B0 | model check. | CBC | proof |
|---|---|---|---|---|---|
| B Compiler | ✓ | ✓ | | | |
| POG and provers | | | | | ✓ |
| ProB | | | ✓ | ✓ | |

**Table 2. Comparison of the tools available for B verification processes**

### 2.1.3 Application: Procedure On-Sight

The Procedure On-Sight, as described in *System Requirements Specification, Chapter 5*, has been modelled in B[5] to show the feasibility of the task and the credibility of the method. This appendix briefly presents the model, then applies the verification processes to this example.

[4]See `http://bware.lri.fr/index.php/BWare_project`

[5]The model is available at `github.com/openETCS/validation/tree/master/VnVUserStories/VnVUserStorySysterel/02-DAS2V/c-ClassicalBModel/ProcedureOnSight`.

**Presentation of the B model** As shown in figure 1, the model is split into three main processing modules, one of which corresponds to the actual on-sight procedure, and the two others being used as data conditioning:

- `os_mode_level`: determines the ETCS level and the mode. Contains the on-sight procedure algorithm,
- `os_consist`: checks data consistency, provides adaptation to the current ETCS level (BTM or radio),
- `os_train_info`: elaborates the location and the speed of the train.

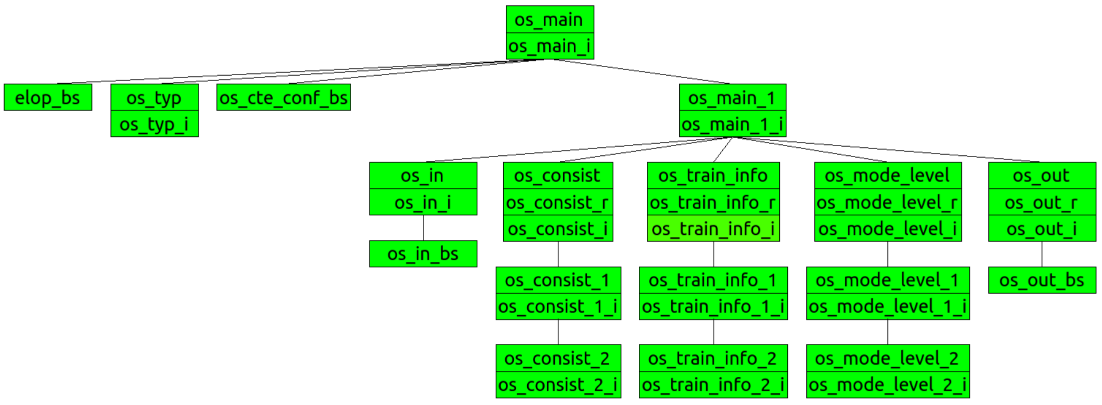


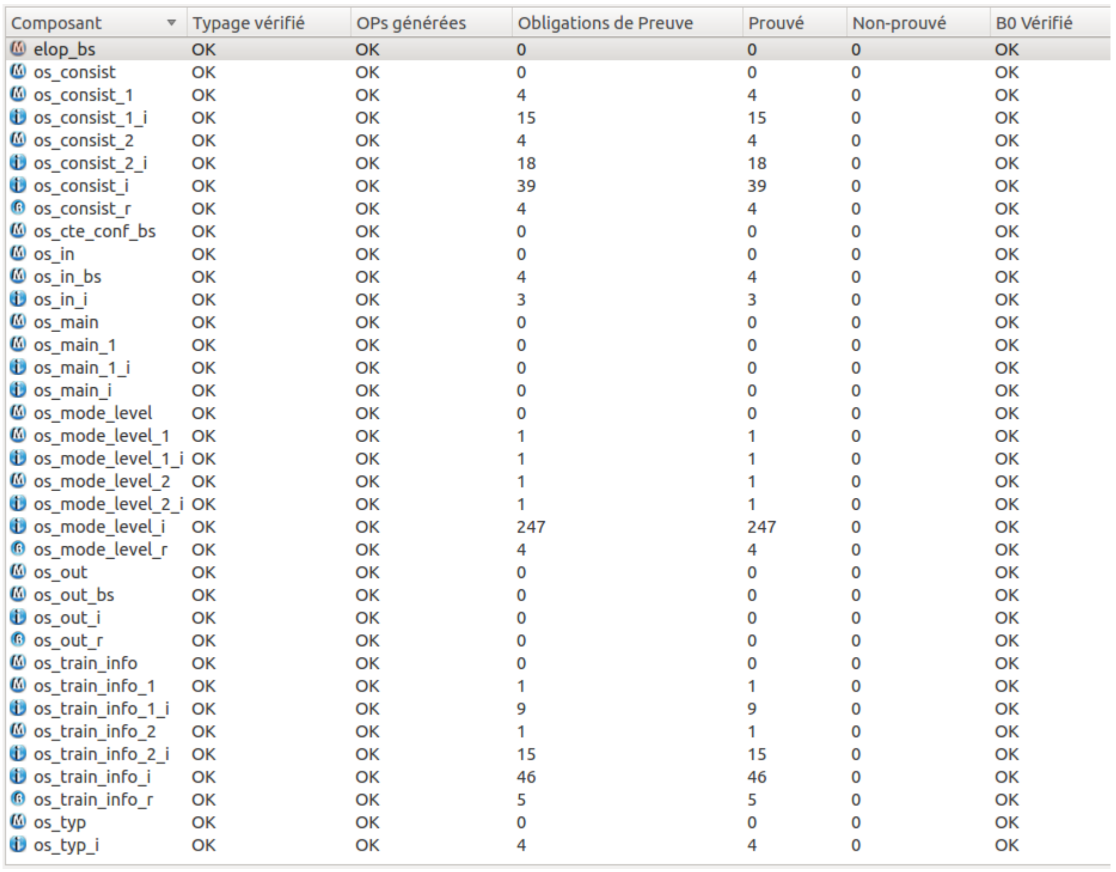
**Figure 1. Architecture of the B model for the Procedure On-Sight example**

These three modules are imported by the main sequencer, `os_main_1`, which calls their respective operations. The main sequencer also imports the input module `os_in`, and the output module `os_out`.
The typing machine, `os_typ`, and the constant machine, `os_cte_conf_bs`, are both imported by `os_main`, the entry point of the software, which also imports `os_main_1`.

This “IMPORTS”-based vertical layout is complemented by a horizontal aspect: the “SEES” clause, which enables a component to access another component’s data. It is possible for a component to see the components to its left, but not to its right. Thus, a cycle-free graph is maintained.

**Model checking results** ProB has been used on the example model and has shown through model checking that no invariant was violated, and no deadlock state was found. However, for some machines, only a minority of states and nodes have been visited (because of infinite sets in particular) and thus no formal conclusion can be drawn.
Additionally, constraint-based checking has also been run and stated, for every operation of every abstract machine, the non-violation of the invariant.

**Formal proof results** Project status illustrated in figure 2 shows the fully-proved model in Atelier B.



| Composant | Typage vérifié | OPs générées | Obligations de Preuve | Prouvé | Non-prouvé | B0 Vérifié |
|---|---|---|---|---|---|---|
| elop_bs | OK | OK | 0 | 0 | 0 | OK |
| os_consist | OK | OK | 0 | 0 | 0 | OK |
| os_consist_1 | OK | OK | 4 | 4 | 0 | OK |
| os_consist_1_i | OK | OK | 15 | 15 | 0 | OK |
| os_consist_2 | OK | OK | 4 | 4 | 0 | OK |
| os_consist_2_i | OK | OK | 18 | 18 | 0 | OK |
| os_consist_i | OK | OK | 39 | 39 | 0 | OK |
| os_consist_r | OK | OK | 4 | 4 | 0 | OK |
| os_cte_conf_bs | OK | OK | 0 | 0 | 0 | OK |
| os_in | OK | OK | 0 | 0 | 0 | OK |
| os_in_bs | OK | OK | 4 | 4 | 0 | OK |
| os_in_i | OK | OK | 3 | 3 | 0 | OK |
| os_main | OK | OK | 0 | 0 | 0 | OK |
| os_main_1 | OK | OK | 0 | 0 | 0 | OK |
| os_main_1_i | OK | OK | 0 | 0 | 0 | OK |
| os_main_i | OK | OK | 0 | 0 | 0 | OK |
| os_mode_level | OK | OK | 0 | 0 | 0 | OK |
| os_mode_level_1 | OK | OK | 1 | 1 | 0 | OK |
| os_mode_level_1_i | OK | OK | 1 | 1 | 0 | OK |
| os_mode_level_2 | OK | OK | 1 | 1 | 0 | OK |
| os_mode_level_2_i | OK | OK | 1 | 1 | 0 | OK |
| os_mode_level_i | OK | OK | 247 | 247 | 0 | OK |
| os_mode_level_r | OK | OK | 4 | 4 | 0 | OK |
| os_out | OK | OK | 0 | 0 | 0 | OK |
| os_out_bs | OK | OK | 0 | 0 | 0 | OK |
| os_out_i | OK | OK | 0 | 0 | 0 | OK |
| os_out_r | OK | OK | 0 | 0 | 0 | OK |
| os_train_info | OK | OK | 0 | 0 | 0 | OK |
| os_train_info_1 | OK | OK | 1 | 1 | 0 | OK |
| os_train_info_1_i | OK | OK | 9 | 9 | 0 | OK |
| os_train_info_2 | OK | OK | 1 | 1 | 0 | OK |
| os_train_info_2_i | OK | OK | 15 | 15 | 0 | OK |
| os_train_info_i | OK | OK | 46 | 46 | 0 | OK |
| os_train_info_r | OK | OK | 5 | 5 | 0 | OK |
| os_typ | OK | OK | 0 | 0 | 0 | OK |
| os_typ_i | OK | OK | 4 | 4 | 0 | OK |

**Figure 2. Overview of the B model in Atelier B, showing type check, B0 check and proof status**

This model was proved almost entirely automatically, using the provers with force 0 and force 1 (on 427 proof obligation generated, only 3 need the interactive prover to define cases). Proving the model results in the certainty of its correctness wrt. the formal specification. In this case, only typing invariants and constraints were expressed, because more complex safety properties have not been identified to be specified as invariant. However, for each function, its behaviour is specified and the implementation is verified according to this specification.

When automatic proof fails, the user must provide a manual proof and uses theories for this purpose. Theories are rules that are used to discharge specific goals.
In this example, the only module for which interactive proof was required is `os_consist_i`. Below is presented a very simple theory (among several others) used for the proof of this component:

$$a < 0 \land 0 \leq b \land 0 < c \Rightarrow a \leq \frac{b}{c} \qquad \textbf{(User theory 1)}$$

This theory is automatically verified by Atelier B and therefore ensures full consistency of the proof.

### 2.1.4 Conclusion

The B method, along with its verification processes and tools, meets the goals and activities of the openETCS project in terms of quality, rigor, safety and credibility.
There is yet to develop open-source POG and build a framework for proving, but this is com-

pensated by the fact that work on the subject is ongoing, and ProB is an effective tool for verification.

## 2.2 Verification and Validation on Event-B models

The Event-B method share the same language than the classical B method. Besides both approaches are based on a pre/post -condition mechanism to describe the evolution of the system. Thus similar verification mechanisms can be defined.

### 2.2.1 Verification Processes Applicable to Event-B

An Event-B model is a formal specification that describes the functional behavior of a system from a global point of view. In general, an Event-B model comprises a set of state variables, parametrized events that can modify these state variables and invariants that describe logical properties thereof. The invariants are in first-order predicate logic and can be discharged using different proof engines, e.g., automatic modern open source SMT solvers and manual predicate provers.

In general, one starts with a rather abstract description of the model which is iteratively refined until the desired level of detail is reached. Event-B supports this refinement by creating the necessary proof obligations that ensure correct refinement in each step, both for behavioral refinement of events as well as for data refinement of state variables.

Thanks to the integration into the Eclipse platform, there are many plug-ins available as extensions. There is a plug-in to use graphical modeling in UML state machines to describe Event-B models. There is a tight connection to the ProR requirements engineering plug-in. To facilitate modeling, there are plug-ins to decompose a model into several sub-models and to facilitate proving by supporting external formal theories.

Together with the Rodin tool, the Event-B approach was developed in the European research projects RODIN (2004–2007) and DEPLOY (2008–2012). Since 2011 it is further developed in the European project ADVANCE.

**Verification of Type Safety** Static type checking is a technique that allows the verification of correct typing for variables at compile / modeling time. It is performed after lexical and syntactic correctness of the Event-B model have been verified. Static type checking prevents all type errors at run-time, which eliminates many possible sources of program errors.

The type system of Event-B is much more expressive than the one of most other languages, as Event-B also allows the usage of dependent types for variables. In this case, the type of a variable is dependent of its value, e.g., one can define the type of all even integers. Event-B can define such types and verify that events respect the correct dependent typing of variables

In Event-B, every new variable gets a type assigned via a typing invariant. Such an invariant is either an explicit type assignment or an implicit one, e.g., by specifying a dependence to another variable which is typed. The integrated type interference can then deduce the static type of the new variable.

Every event that changes the value of a variable via substitution must also respect the variable typing. For each event that modifies a variable, proof obligations are created that ensure this in a rigorous formal way.

In almost all cases, the proof obligations for type verification are discharged automatically by the Rodin provers.

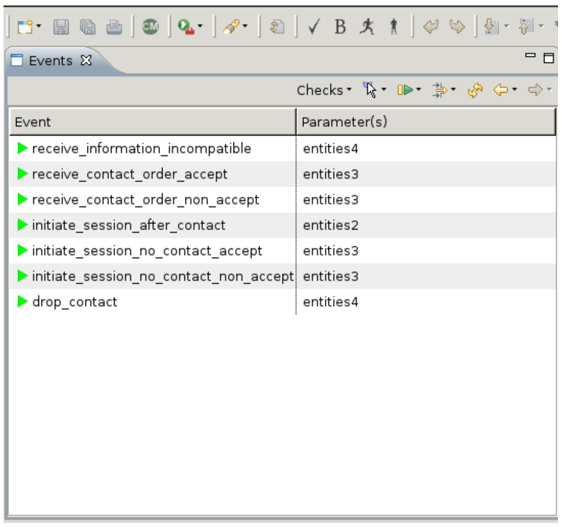
**Verification of Well-Definedness** After type checking, one or more well-definedness (WD) proof obligations are created. This ensures that the expression has a unique meaning and prevents the usage of expressions that make no sense or are ambiguous.

One prominent example for WD proof obligations in Event-B is the cardinality of sets. The set of natural numbers $\mathbb{N}$ has countable infinitely many elements, exactly as many as the set of all even natural numbers $\mathbb{N}_2 := \{2 \cdot n \mid n \in \mathbb{N}\}$. This means that both sets have the same cardinality, although $\mathbb{N}_2$ is a strict subset of $\mathbb{N}$.

Therefore, while sets of countable infinite cardinality can be used without any problem in Event-B models, the usage of cardinality of a set requires the set to be of finite size which gets verified by an appropriate WD proof obligation.

In almost all cases, the proof obligations for well-definedness are discharged automatically by the Rodin provers.

**Model Simulation** A correctly typed Event-B model can be simulated or animated using different plug-ins like AnimB or ProB. At each step, one of the activated events can be executed and if applicable parameters for that event can be defined. This allows for stepping through the formal model, observing the state variables and the invariants. Using model animation, it is possible to validate the correct functioning of the model.



| Event | Parameter(s) |
|---|---|
| ▶ receive_information_incompatible | entities4 |
| ▶ receive_contact_order_accept | entities3 |
| ▶ receive_contact_order_non_accept | entities3 |
| ▶ initiate_session_after_contact | entities2 |
| ▶ initiate_session_no_contact_accept | entities3 |
| ▶ initiate_session_no_contact_non_accept | entities3 |
| ▶ drop_contact | entities4 |

**Figure 3. ProB Model Animation**

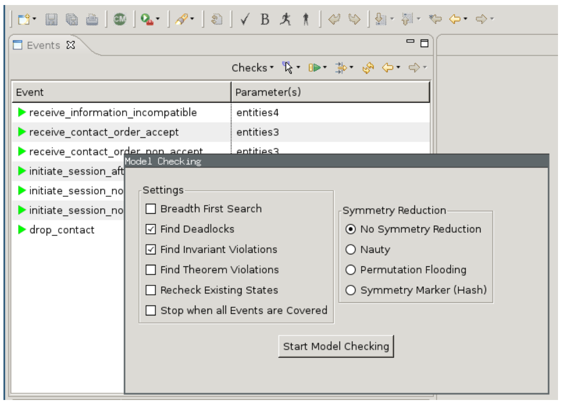
Figure 3 shows a ProB simulation session. The activated events are marked green, clicking on them allows for selection of parameters and to execute the events with the chosen parameters.

**Model-Checking of Predicates** Model-checking is a static analysis of the semantics of a model. In general, a model-checker will create a representation of the whole possible state space of a model and verify logic properties on this state space. There are many different possibilities for properties that are verified by model-checkers, some are listed here:

- **Equivalence Checking** The equivalence between two models is verified given a certain equivalence relation. Often, a specification is compared to its (distributed) implementation using bisimulation modulo some reduction techniques, e.g., only the externally observable behavior is compared and the internal details of the different systems are ignored.
- **Deadlock Freeness** A deadlock represents a state where the system that the system cannot leave as no event is enabled. For a reactive system this is always an unwanted state that must be avoided.
- **Temporal Properties** The evolution of the system over time is analyzed, i.e., the admissible event sequences that can lead to different states. Roughly, temporal properties comprise *safety properties* which describe a set of states that should never be reached and *liveness properties* which describe states that should always be reachable. There are different temporal logic languages, like LTL and CTL, which allow to describe temporal properties of systems.

In general, model-checkers suffer from the state space explosion problem. This means that creating the whole state space becomes often infeasible due to memory limitations. In general it is also not possible to model-check systems with infinite state space, like many Event-B models.

In practice, tools like ProB which allow for model-checking of Event-B models, limit the size of the possible values for variables to a finite subset. While this means that a complete proof is not possible, it allows for fully automatic error detection in the model. For any violated property or a deadlock, ProB provides a counterexample that can be analyzed and therefore allows for correction of the associated modeling problem.



| Event | Parameter(s) |
|---|---|
| receive_information_incompatible | entities4 |
| receive_contact_order_accept | entities3 |
| receive_contact_order_non_accept | entities3 |
| initiate_session_aft | |
| initiate_session_no | |
| initiate_session_no | |
| drop_contact | |

Model Checking

Settings
- ☐ Breadth First Search
- ☑ Find Deadlocks
- ☑ Find Invariant Violations
- ☐ Find Theorem Violations
- ☐ Recheck Existing States
- ☐ Stop when all Events are Covered

Symmetry Reduction
- ◉ No Symmetry Reduction
- ○ Nauty
- ○ Permutation Flooding
- ○ Symmetry Marker (Hash)

Start Model Checking

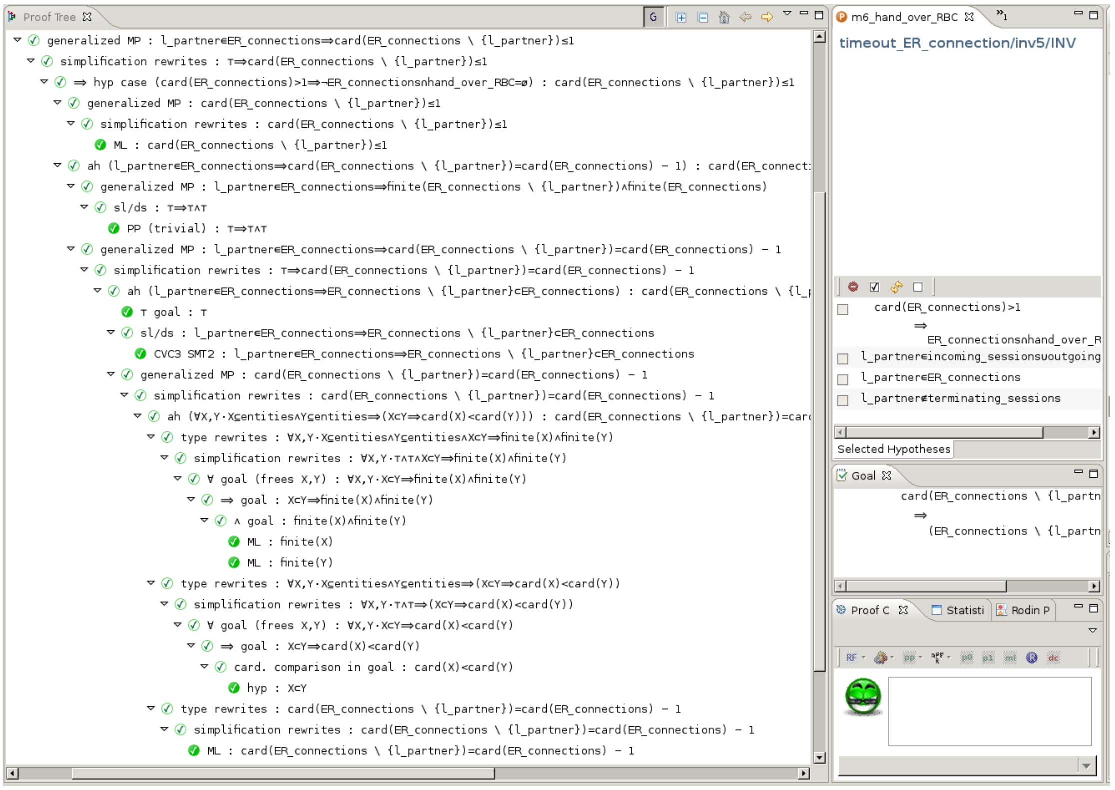
**Figure 4. Model-Checking for Deadlocks**

Figure 4 shows the model checking dialog of the Rodin plug-in for ProB. The currently selected options would check for deadlocks, i.e., a sequence of events that leads to a situation where no event can be selected anymore.

**Formal Proof of Predicates** Formal proof techniques provide a much more powerful way to verify predicates than model-checking. Instead of the creation of the full state-space, they use a proof calculus to iteratively simplify predicates and to reduce them onto known lemmas or axioms, thus discharging them.

In contrast to model-checking, formal proof is applicable to models of infinite size and can cope with undecidable problems. Although this means that there sometimes will be a manual step in a proof, there are many automated tools that support formal proofs and can often discharge proof obligations without any manual intervention.

The Rodin platform natively supports manual construction of formal proofs by allowing easy access and manipulation of the proof tree and predicate hypotheses. It also provides access to different automated provers, i.e., the free of charge AtelierB provers, an open source SMT plug-in that supports several solvers[6] as well as an open source plug-in that connects Rodin to the Isabelle/HOL proof assistant.



```
generalized MP : l_partner∈ER_connections⇒card(ER_connections \ {l_partner})≤1
  simplification rewrites : ⊤⇒card(ER_connections \ {l_partner})≤1
    ⇒ hyp case (card(ER_connections)>1⇒¬ER_connections∩hand_over_RBC=∅) : card(ER_connections \ {l_partner})≤1
      generalized MP : card(ER_connections \ {l_partner})≤1
        simplification rewrites : card(ER_connections \ {l_partner})≤1
          ML : card(ER_connections \ {l_partner})≤1
      ah (l_partner∈ER_connections⇒card(ER_connections \ {l_partner})=card(ER_connections) − 1) : card(ER_connect
        generalized MP : l_partner∈ER_connections⇒finite(ER_connections \ {l_partner})∧finite(ER_connections)
          sl/ds : ⊤⇒⊤∧⊤
            PP (trivial) : ⊤⇒⊤∧⊤
        generalized MP : l_partner∈ER_connections⇒card(ER_connections \ {l_partner})=card(ER_connections) − 1
          simplification rewrites : ⊤⇒card(ER_connections \ {l_partner})=card(ER_connections) − 1
            ah (l_partner∈ER_connections⇒ER_connections \ {l_partner}⊂ER_connections) : card(ER_connections \ {l_
              ⊤ goal : ⊤
              sl/ds : l_partner∈ER_connections⇒ER_connections \ {l_partner}⊂ER_connections
                CVC3 SMT2 : l_partner∈ER_connections⇒ER_connections \ {l_partner}⊂ER_connections
              generalized MP : card(ER_connections \ {l_partner})=card(ER_connections) − 1
                simplification rewrites : card(ER_connections \ {l_partner})=card(ER_connections) − 1
                  ah (∀X,Y·X⊆entities∧Y⊆entities⇒(X⊂Y⇒card(X)<card(Y))) : card(ER_connections \ {l_partner})=card
                    type rewrites : ∀X,Y·X⊆entities∧Y⊆entities∧X⊂Y⇒finite(X)∧finite(Y)
                      simplification rewrites : ∀X,Y·⊤∧⊤∧X⊂Y⇒finite(X)∧finite(Y)
                        ∀ goal (frees X,Y) : ∀X,Y·X⊂Y⇒finite(X)∧finite(Y)
                          ⇒ goal : X⊂Y⇒finite(X)∧finite(Y)
                            ∧ goal : finite(X)∧finite(Y)
                              ML : finite(X)
                              ML : finite(Y)
                    type rewrites : ∀X,Y·X⊆entities∧Y⊆entities⇒(X⊂Y⇒card(X)<card(Y))
                      simplification rewrites : ∀X,Y·⊤∧⊤⇒(X⊂Y⇒card(X)<card(Y))
                        ∀ goal (frees X,Y) : ∀X,Y·X⊂Y⇒card(X)<card(Y)
                          ⇒ goal : X⊂Y⇒card(X)<card(Y)
                            card. comparison in goal : card(X)<card(Y)
                              hyp : X⊂Y
                    type rewrites : card(ER_connections \ {l_partner})=card(ER_connections) − 1
                      simplification rewrites : card(ER_connections \ {l_partner})=card(ER_connections) − 1
                        ML : card(ER_connections \ {l_partner})=card(ER_connections) − 1
```

```
timeout_ER_connection/inv5/INV

card(ER_connections)>1
    ⇒
      ER_connections∩hand_over_R
l_partner∈incoming_sessions∪outgoing
l_partner∈ER_connections
l_partner∉terminating_sessions
```

```
card(ER_connections \ {l_partn
  ⇒
    (ER_connections \ {l_partn
```

Figure 5. Rodin Proof Tree

Figure 5 shows a part of a Rodin proof tree for an invariant. Its green color signals that the proof is finished, at each node in the tree, the applied proof rule is shown. This allows for easy inspection of the proofs and allows both for humans and for machines to verify the correctness of the proof steps.

**Verification of Refinement Correctness** Rodin provides extensive support for a top-down development approach and allows for an iterative refinement of models. The model is developed using different levels of detail, starting from a rather abstract view, refining the details where necessary or desired. This refinement process can either be applied to the events or the variables.

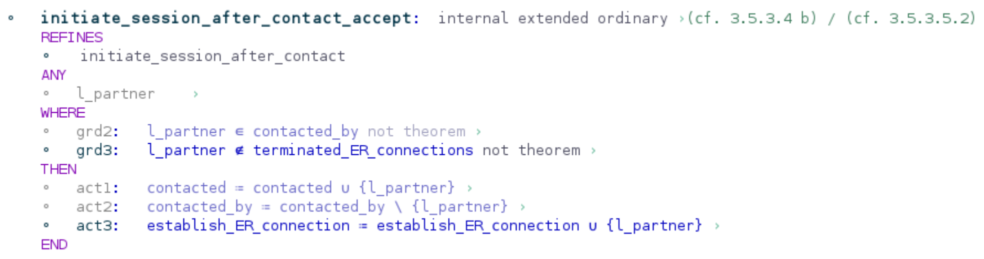
### 2.2.2 Data Refinement

[6]supported open source SMT solvers include: verIT, Alt-Ergo, CVC3, Z3

In general, a data refinement replaces a variable with another one, or multiple other ones. For example a Boolean variable in the abstract model is replaces by an enumeration with different possible values. To ensure a correct refinement, one has to manually supply a “gluing” invariant that describes the connection of the refined and the abstract variable. For example one subset of the possible values for the enumeration in the refined model would correspond to a value of “True” in the abstract model, the remaining values of the enumeration to a value “False”. The abstract variable is then deleted from the refined model, and the necessary proof obligations are created automatically by Rodin.

### 2.2.3 Code Refinement

For event (or code) refinement, Rodin automatically creates the necessary proof obligations that ensure that the abstract system is correctly refined by the more detailed model. This includes the verification that each refining event only modifies variables that are also modified by the abstract event and that the modification is equivalent. It also includes verification of guard strengthening, i.e., the guards of a refining event must be at least as constraining as of the refined abstract event. A common code refinement is to split an event in several more specialized ones, where the additional guards ensure mutual exclusion of the activation conditions.



```
∘  initiate_session_after_contact_accept:  internal extended ordinary ›(cf. 3.5.3.4 b) / (cf. 3.5.3.5.2)
   REFINES
   ∘   initiate_session_after_contact
   ANY
   ∘   l_partner    ›
   WHERE
   ∘   grd2:   l_partner ∈ contacted_by not theorem ›
   ∘   grd3:   l_partner ∉ terminated_ER_connections not theorem ›
   THEN
   ∘   act1:   contacted ≔ contacted ∪ {l_partner} ›
   ∘   act2:   contacted_by ≔ contacted_by \ {l_partner} ›
   ∘   act3:   establish_ER_connection ≔ establish_ER_connection ∪ {l_partner} ›
   END
```

**Figure 6. Event Refinement**

Figure 6 shows a refining event with guard strengthening and an additional variable that is modified. The pale blue colored guards and actions are derived from the refined event, the darker colored guard and action are the additional ones for the refining event.

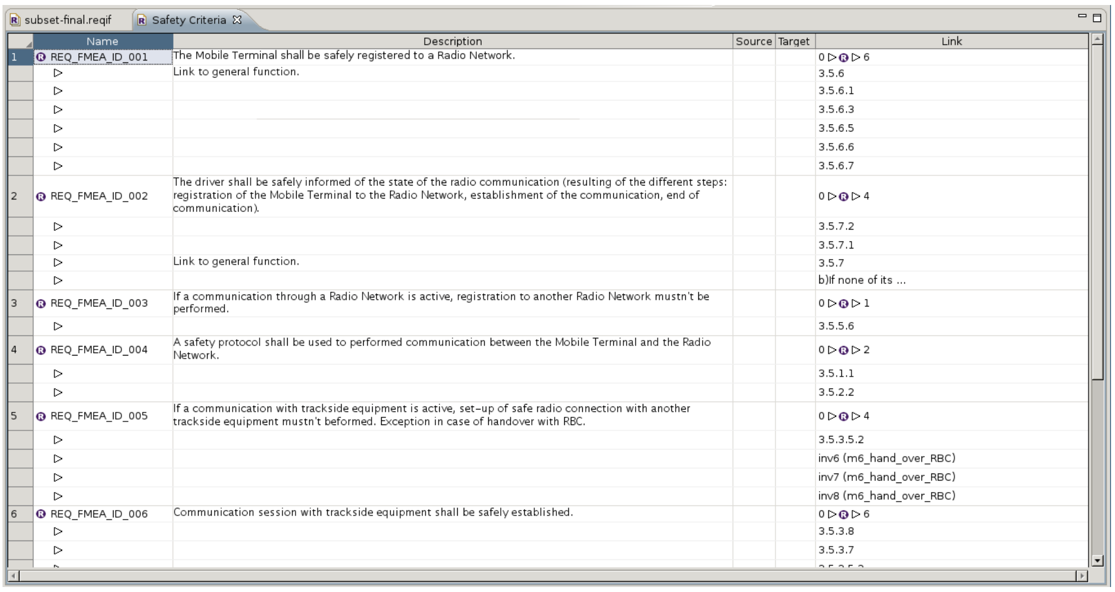
**Verification of Design Step Requirements** This section reports on the verification activities of the correct implementation of the design step requirements. The goal of the activity is to establish a correct formal representation of the design step requirements.

The activity is described in the Verification and Validation Plan [5]. In short, it consists of formalizing and proving the identified requirements of a preceding phase, to ensure their correct implementation of the requirements. To achieve this, we use the direct connection between the Event-B model and the ProR based on an EMF model of the Event-B model.

**Verification of Safety Requirements** A safety analysis identifies additional requirements which guarantee the safety of a system. It must be verified that the system model correctly implements these non-functional requirements. Not every safety requirement is applicable on the system development level. Many are on the implementation level, e.g., they demand that certain safety-critical functions are done in a redundant way to reduce the risk of malfunctioning or loss of that function.

In the safety analysis [6], a list of safety requirements was identified using an FMEA analysis of the communication system. A ReqIf file captures all these safety requirements within ProR, the concerned functional requirements are traced in the ReqIf file for Subset-026 chapter 3.5.

Each of the safety requirements is examined for applicability in the system level model, the identified ones are formalized. Most often, the safety requirements are represented as one or more additional invariants in the system model. These invariants are linked to the ReqIf file that describes the safety requirements, ensuring traceability in the model.



| | Name | Description | Source | Target | Link |
|---|---|---|---|---|---|
| 1 | REQ_FMEA_ID_001 | The Mobile Terminal shall be safely registered to a Radio Network. | | | 0 ▷ R ▷ 6 |
| | ▷ | Link to general function. | | | 3.5.6 |
| | ▷ | | | | 3.5.6.1 |
| | ▷ | | | | 3.5.6.3 |
| | ▷ | | | | 3.5.6.5 |
| | ▷ | | | | 3.5.6.6 |
| | ▷ | | | | 3.5.6.7 |
| 2 | REQ_FMEA_ID_002 | The driver shall be safely informed of the state of the radio communication (resulting of the different steps: registration of the Mobile Terminal to the Radio Network, establishment of the communication, end of communication). | | | 0 ▷ R ▷ 4 |
| | ▷ | | | | 3.5.7.2 |
| | ▷ | | | | 3.5.7.1 |
| | ▷ | Link to general function. | | | 3.5.7 |
| | ▷ | | | | b)If none of its ... |
| 3 | REQ_FMEA_ID_003 | If a communication through a Radio Network is active, registration to another Radio Network mustn't be performed. | | | 0 ▷ R ▷ 1 |
| | ▷ | | | | 3.5.5.6 |
| 4 | REQ_FMEA_ID_004 | A safety protocol shall be used to performed communication between the Mobile Terminal and the Radio Network. | | | 0 ▷ R ▷ 2 |
| | ▷ | | | | 3.5.1.1 |
| | ▷ | | | | 3.5.2.2 |
| 5 | REQ_FMEA_ID_005 | If a communication with trackside equipment is active, set–up of safe radio connection with another trackside equipment mustn't beformed. Exception in case of handover with RBC. | | | 0 ▷ R ▷ 4 |
| | ▷ | | | | 3.5.3.5.2 |
| | ▷ | | | | inv6 (m6_hand_over_RBC) |
| | ▷ | | | | inv7 (m6_hand_over_RBC) |
| | ▷ | | | | inv8 (m6_hand_over_RBC) |
| 6 | REQ_FMEA_ID_006 | Communication session with trackside equipment shall be safely established. | | | 0 ▷ R ▷ 6 |
| | ▷ | | | | 3.5.3.8 |
| | ▷ | | | | 3.5.3.7 |

**Figure 7. Safety Requirements**

Figure 7 shows a ReqIf document in Rodin (via the ProR plug-in) which holds the safety requirements defined by the safety analysis. For each requirement, there are references to the concerned elements of Subset-026 and to Event-B elements where applicable, e.g., REQ_FMEA_ID_005 which is linked to the invariants, inv6, inv7 and inv8.

**Verification of Requirements Coverage** This section reports on the verification activities of the coverage of the design step requirements. The goal of the activity is to establish the coverage degree of the formal representation of the design step requirements.

The activity is described in the Verification and Validation Plan [5]. In short, in consists of analyzing the coverage of the identified requirements of a preceding phase, to ensure their completeness of implementation of the requirements wrt. the refinement level of the model. To achieve this, we use the direct connection of the Event-B model with the ProR based on an EMF model of the Event-B model.

### 2.2.4 Object of Verification

The object of verification is the Event-B model for the communication establishing at `https://github.com/openETCS/model-evaluation/tree/master/model/Event_B_Systerel/Subset_026_comm_session`. It is from the strictly formal modeling phase and represents the communication session management of the OBU.

**Available Specification** The model implements the requirements for the communication session management as described in Subset-026 chapter 3.5.

This section describes the establishing, maintaining and termination of a communication session of the OBU with on-track systems.

**Goals** One goal is the development of a strictly formal, fully proven model of the communication session management and to provide evidence of covering the necessary requirements of Subset-026 as well as proving correctness of the model wrt. the requirements and attaining a good coverage of the model wrt. the requirements.

The second goal is to correctly implement the applicable safety requirements identified by the safety analysis. Both functional and safety requirements should be traced in the model and a requirement document in a standardized format.

The formal model will represent the described functionality on the system level, the correct functioning can be validated by step-wise simulation and model-checking of deadlock-freeness.

**Method/Approach** At first, the basic functionality described in the chapter 3.5 that are identified. These serve as basis for a first abstract model, which is refined iteratively, adding the desired level of detail. The elements of Subset-026 are traced using links from Event-B to the ProR file in ReqIf format. Requirements are formalized as invariants and proven where applicable.

**Means** The means used are:

- open source Rodin tool (`http://www.event-b.org/`), including plug-ins (for details see `https://github.com/openETCS/model-evaluation/blob/master/model/Event_B_Systerel/Subset_026_comm_session/latex/subset_3_5.pdf`)
- ProR requirements modeling tool `http://www.pror.org`
- open source ProB model checker and B model simulator `http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page`
- open source CVC3 (`http://www.cs.nyu.edu/acsys/cvc3/`), verIT (`www.verit-solver.org`) and Alt-Ergo (`http://alt-ergo.lri.fr`) SMT solvers

**Results**

- The result is a fully formal model of the communication session management as described in chapter 3.5 of Subset-026.
- Each implemented element of this section is linked to the ProR requirements file, both specification elements that describe how something has to be done, as well as requirements that describe what must be achieved.
- The model can be simulated / animated, either with the AnimB or the ProB plug-in, validating the functional capabilities.

- The safety requirements are formalized as invariants in predicate logic, their proofs are for the most part fully automatic.
- It was found that while the Subset-026 communication management explicitly allows multiple communication partners (see RBC handover), there is no explicit limit of established communication connections given in chapter 3.5.
- A complete covering of the elements of Subset-026 was not realized, e.g., there is a representation of the contents of a message, but its explicit format is not implemented. This is considered an implementation detail without influence for a system level analysis. In general, Event-B models will not be refined up to the implementation level.

**Summary** The created fully formal functional model allows for formalization and proof of Subset-026 requirements. The integration of Rodin into Eclipse provides easy access to extensions like the ProR requirements tool which allows for validation of coverage of requirements.

The integration of various provers, in particular the SMT plug-in automates a large part of the formal verification. For the model of the communication management, from 382 non-trivial[7] proof obligations, only 12, i.e., 3.2% require any manual intervention.

**Evidence produced** The formal Event-B model, including a ReqIf document for chapter 3.5 of Subset-026 and a pdf documentation of the model can be found at `https://github.com/openETCS/model-evaluation/tree/master/model/Event_B_Systerel/Subset_026_comm_session`

### 2.2.5 Conclusions/Lessons learned

Having an abstract formal model of the implemented functionality which can be simulated, allows for interesting insights into the overall functioning of a system. Formalized requirements are very helpful in both the identification of ambiguous requirements and in their clarification.

The elements of Subset-026 are of very different nature. Some describe rather low-level specification details, other describe "real" requirements. Without an analysis as done with this Event-B model, it can be difficult to decide which elements must be considered on a system level analysis and which on the lower implementation level.

### 2.2.6 Future Activities

For other sections of Subset-026, that describe a functionality in a way that can be captured in an iteratively refined model and which has interesting requirements on a rather high level, creating an Event-B model can provide insight into the functioning, identify ambiguous or erroneous elements in the specification and can provide the basis for logical pre- and post conditions of the later implementation.

## 2.3 Verification processes applicable to a SCADE model

The verifier shall be independent and shall neither be Requirements Manager, Designer nor Implementer as defined in the safety standards EN 50128 v2011.

[7]many WD proof obligations are so trivial that they will not be shown in Rodin

The input documents needed are all the necessary System and Software Documentation used for the SCADE design activity and all the documentation produced during this phase, such as the SCADE Design Description, the SCADE Design Test Specification and the SCADE Design Test Report.

### 2.3.1 Respect of modelling rules

Syntactic rules of SCADE language are verified with the Quick Check tool available in the publisher. If an error is detected it must be corrected or justified in the SCADE Design Description document by the designer. The verifier shall ensure that no error remains or the justification associated is correct.

For specific modelling rules the verification has to be made manually by the verifier and described in the Verification Report. A grid of verification may be created in order to prove the compliance of the model with the rules. On some cases, dedicated tools can be developed.

Some modelling rules and constraints on Scade language can be defined and justified according CENELEC standard. Then these rules can be verified.

### 2.3.2 Specification traceability check

The verification of the compliance of the SCADE model with each requirement has to be made manually, by the verifier.

The Scade model shall be correct according to the informal requirements and the informal specification shall be completely covered : each specification requirement must be traced in the SCADE model. The specification requirements which are not covered by the SCADE model must be listed and justified in the SCADE Design Description document by the designer.

### 2.3.3 Testing and Validation of the model

The verifier shall control the activity of software testing performed by the tester.

The software testing uses the Model Test Coverage (MTC) and the Generic Qualified Testing Environment (QTE) tools from SCADE. Five steps are performed.

- Establish the Test Specification document.
- Writing scenarios in order to test the different functions independently.
- Running scenarios on the SCADE model.
- Extraction and analysis of results and the associated coverage (nodes, branches, branch conditions,...).
- Establish the Test Report.

### 2.3.4 Results

All these different verifications activities shall be described in the Verification and Validation Plan, and their results shall be record in a Verification Report. Each disparity must be corrected or justified.

**Verification report content** The verifier shall produce a Verification Report containing the proof of the compliance of the SCADE model. It shall include the following points:

- the identity, version and configuration of SCADE model;
- the verifier name;
- the goal of the Verification Report;
- the result of each verification process with:
  - items which do not conform to the specifications;
  - components, data, structures and algorithms poorly adapted to the problem;
  - detected errors or deficiencies.
- the fulfilment of, or deviation from, the Software Verification Plan;
- assumptions if any;
- a summary of the verification results.

### 2.3.5 Conclusion

The use of SCADE with its verification processes is compliant with the CENELEC norm but as it is not developed as open-source it is not compliant with the goal of openETCS project.

# 3 LAAS and INPT: Verification of the Ceiling Speed Monitoring

This section reports the verification activity of the Ceiling Speed Monitoring (CSM) function provided by the University of Bremen using the Tina model-checking toolbox http://projects.laas.fr/tina/. The goal of this activity is to use an automatic transformation from SysML to Time Petri Net (TPN) to this model and check several temporal logic formulas on the resulting system.

## 3.1 Object of verification

We study the SysML model of CSM function using the Tina model-checking toolbox. We base our analysis on a model provided by the University of Bremen that was slightly extended with information on the environment of the system. The same function was studied by the University of Rostock using a software testing approach.

The cornerstone of our approach is an automatic transformation from SysML models to TPN models. We can then use the resulting formal model to check several temporal logic formulas.

## 3.2 Available specification

The CSM function supervises the observance of the maximal speed allowed according to the current most restrictive speed profile (MRSP). The model was edited, and later extended, using Papyrus. The specification of the system under test is described in [7] and available here: `https://github.com/openETCS/validation/tree/master/VnVUserStories/VnVUserStoryUniBremen`.

To provide executable models for the CSM function, an environment model needs to be defined; mainly the possible actions on the current speed of the train resulting from acceleration or deceleration orders.

The combination of a test environment model and an optional test driver model provides a deterministic model.

### 3.2.1 Description of the Environment Model

The environment model is defined in the TestEnvironment block. It is described using a state machine diagram as shown in Figure 8.

The system under test is activated with an initial speed (SimulatedTrainSpeed) and may enter into the composite state Running. The Running state encapsulates three possible behavior. Either the speed remains unchanged (state Normal), or the train accelerates (state Acc), or the train decelerates (state Dec). The guards defined on the transitions found inside the composite state Running are described in Table 3. The DriveCommand is the command sent by the driver which contains three modes: keeping speed (DriveCommand = 0), acceleration (DriveCommand = 1) and Deceleration (DriveCommand = 2).

**Table 3. Transitions between Running States**

| To \ From | Normal | Acc | Dec |
|---|---|---|---|
| Normal | | DriveCommand != 1 | DriveCommand != 2 |
| Acc | DriveCommand == 1 | | |
| Dec | DriveCommand == 2 | | |

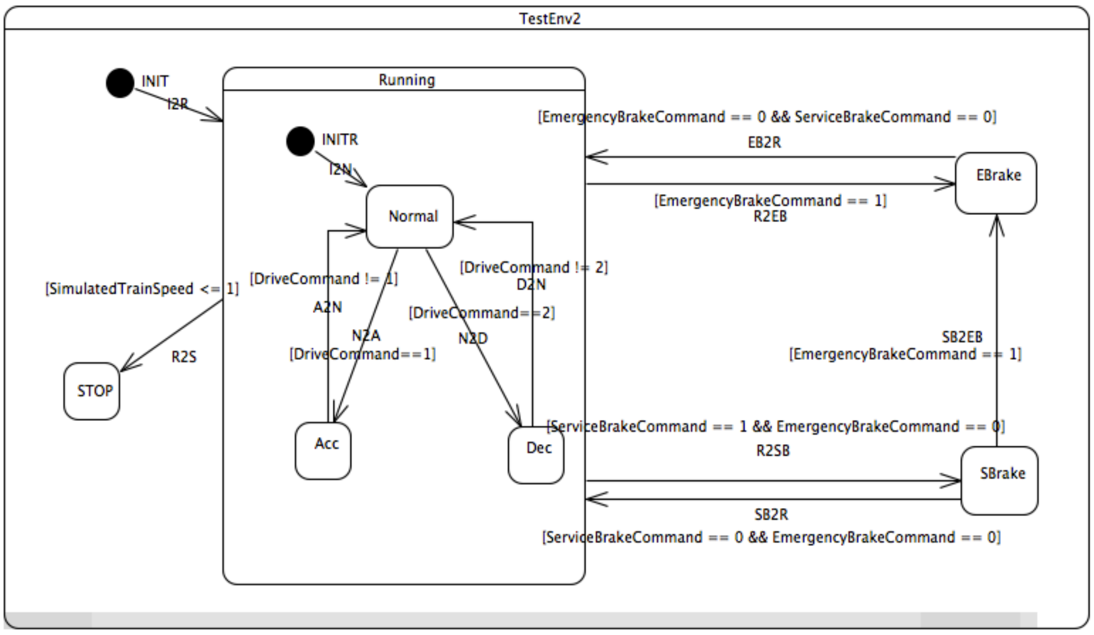

TestEnv2
INIT
I2R
Running
INITR
I2N
Normal
[DriveCommand != 1]
A2N
N2A
[DriveCommand==1]
[DriveCommand != 2]
D2N
[DriveCommand==2]
N2D
Acc
Dec
[SimulatedTrainSpeed <= 1]
R2S
STOP
[EmergencyBrakeCommand == 0 && ServiceBrakeCommand == 0]
EB2R
EBrake
[EmergencyBrakeCommand == 1]
R2EB
SB2EB
[EmergencyBrakeCommand == 1]
[ServiceBrakeCommand == 1 && EmergencyBrakeCommand == 0]
R2SB
SBrake
SB2R
[ServiceBrakeCommand == 0 && EmergencyBrakeCommand == 0]

**Figure 8. CSM Test Environment Model**

The actions on each transition (the "do behaviors") are described in Table 4. At the moment we use dummy values for the acceleration and braking parameters of the train. We have chosen a fix acceleration of 2 km/h per 100 units of time (t.u.) and a constant braking factor of 2 km/h per 400 t.u.

**Table 4. Do Behaviors in Running States**

| | |
|---|---|
| Normal | |
| Acc | SimulatedTrainSpeed = SimulatedTrainSpeed + 2; |
| Dec | SimulatedTrainSpeed = SimulatedTrainSpeed - 2; |

The effect of the environment on the system is described by the transitions between states Running, EBrake (emergency brake), SBrake (service brake) and STOP (simulatedTrain Speed <= 1). The transition guards between environment states are described in Table 5. The transitions are controlled by the commands EmergencyBrakeCommand and ServiceBrakeCommand generated from the train control system.

**Table 5. Transitions between Environment States**

| To \ From | Running | EBrake | SBrake | STOP |
|---|---|---|---|---|
| Running | | ServiceBrakeCommand == 0 && EmergencyBrakeCommand == 0 | ServiceBrakeCommand == 0 && EmergencyBrakeCommand == 0 | |
| EBrake | EmergencyBrakeCommand == 1 | | ServiceBrakeCommand == 0 && EmergencyBrakeCommand == 1 | |
| SBrake | ServiceBrakeCommand == 1 && EmergencyBrakeCommand == 0 | ServiceBrakeCommand == 1 && EmergencyBrakeCommand == 0 | | |
| STOP | SimulatedTrainSpeed <= 1 | | | |

The do behaviors in environment states are described in Table 6. The deceleration of emergency brake is set at 10 km/h per 200 t.u.. The deceleration of service brake is set at 5 km/h per 200 t.u..

**Table 6. Do Behaviors in Environment States**

| | |
|---|---|
| Running | |
| EBrake | SimulatedTrainSpeed = SimulatedTrainSpeed - 10; |
| SBrake | SimulatedTrainSpeed = SimulatedTrainSpeed - 5; |
| STOP | |

### 3.2.2 Description of the Driver Model

In addition to the model of the train behavior we have added a model of the driver (and of the track description) that can be used to force a particular scenario. In this context, a scenario is a timed annotated sequence of acceleration and deceleration orders. One possible scenario can be obtained using the model defined in the TestDriver block of Fig. 9. This test describes a situation where the driver accelerates for a time T1 (DriveCommand = 1) before decelerating for a time T2.

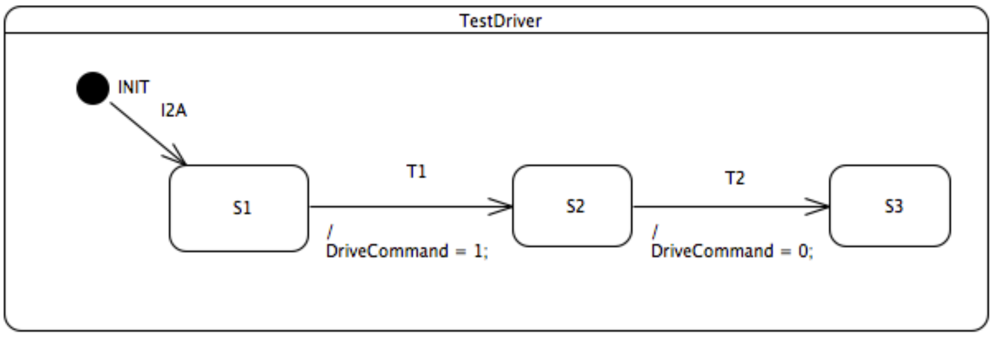


**Figure 9. Test Driver Model**

## 3.3 Detailed verification plan

### 3.3.1 Goals

Our main goal is to reuse the existing CSM model and extend the environment model with the driver model. These SysML models are then transformed to TPN models to validate the specification by model-checking.

### 3.3.2 Method/Approach

We first provide some background information on the transformation from SysML to TPN used in our study. This transformation is based on a mapping from UML (Unified Modeling Language) to TPN described in the PhD thesis of Ning Ge [8]. The transformation can also take into account real-time properties defined using the MARTE profile (Modeling and Analysis of Real Time and Embedded systems).

By nature, UML is intended to be a general purpose modeling language and, as such, it integrates different modeling viewpoints through the definition of a large class of diagram elements. In the work of Ge, they select a core subset of UML diagrams and diagram elements for modeling real-time software architecture and behavior, and focus on the semantic mapping from the UML model to the verification model.

We briefly describe the different elements supported in our translation.

**Architecture Model** The purpose of architecture model is to connect different sub-system behavior models and create a system-level model, by means of communication media. The objective of the mapping is to replace each architecture model's entities by its relevant behavior model. We rely on the composite structure diagram as the architecture model. Composite structure diagrams specify the internal structure of a class, including its interaction points to other parts of the system, and the architecture of all parts managed by this class. They are used to explore run-time instances of interconnected instances collaborating over communications links.

**Behavioral Model** The mapping semantics for behavioral model covers both activity and state machine diagrams.

Activity diagrams express the coordination between lower-level behaviors using constraints on the possible sequence of actions. In this context, actions can be triggered because other actions finish executing; because objects and resources become available; or because external events occur. The main elements in UML activity diagram behavior model are control nodes, actions, objects, and connection elements.

**Principles of Semantic Mapping**

The mapping from UML-MARTE to TPN preserves the semantics of the input language. A particularity of the approach is that, for efficiency reason, the transformation is driven by the set of real-time properties that should be checked on the resulting model. For instance, in order to reduce the size of the state space explored during the verification phase, the behavior of some elements irrelevant to the target property can be abstracted. The transformation conforms to the following principles:

- The resulting TPN models should be easy to analyze, meaning that the semantics mapping should allow the use of high-level abstraction methods during model-checking.
- In order to keep the transformation simple, we use a compositional approach where the resulting system is obtained by composing the interpretation of all its elements. Then, to optimize the result, we apply a state space reduction phase that eliminates the elements irrelevant to the verification.

### 3.3.3 Means

Instead of the thirteen diagrams available in UML 2, the SysML includes only nine diagrams, including:

- the Block Definition Diagram (BDD), replacing the UML 2 class diagram
- the Internal Block Definition Diagram (IBDD), replacing the UML 2 composite structure diagram
- the Parameter diagram, a SysML extension to analyze critical system parameters
- the Package diagram remained unchanged.

The behavior diagrams includes:

- the activity diagram, slightly modified from UML 2

- the sequence, state machine, and use case diagrams remain unchanged.

The requirement diagram is a SysML extension to describe functional, performance, and interface requirements.

In order to reuse the existing transformation form UML to TPN [8] to build a mapping from SysML, we have redefined the mapping semantics for the block diagram as structure models. The mapping semantics for the activity and state machine diagrams are left unchanged. Some of the semantics mapping have also been modified in order to take into account some of the modeling convention adopted in the OpenETCS project.

Also, the target model is now an extension of TPN with priorities and typed variables called TTS, for Time Transition System. The data handling ability of TTS is used to model the guards and actions on integer and float variables found in a SysML diagram.

## 3.4 Results

We provide two verification scenarios for testing the formal system obtained from the transformation of the CSM model. Each scenario is available as a TTS "file" (actually a folder called tpn.tts) inside the 05-Work folder.

**Scenario 1** Scenario 1 includes the following initial values for the parameters:

- SimulatedTrainSpeed = 110
- V_mrsp = 120
- SBAvailable = true
- DriveCommand = 1. The initial drive command is acceleration.
- T1 transition in Driver model has an effect behavior DriveCommand = 1. The time duration for the initial behavior (acceleration) is 20000.
- T2 transition in Driver model has an effect behavior DriveCommand = 0. The time duration for the behavior before keeping speed (acceleration) is 10000.

We give in the table below the number of reachable states (or markings) of the resulting TPN. A marking is defined by a particular value for each system variable and for each internal state of the blocks. This gives a rough idea of the complexity of checking reachability properties on the system. "Classes" take into account timing constraints on top of the markings; hence there is always more classes than markings. The generation of the whole state space takes for Scenario 1 takes less than 24 seconds (system time: 23.350s).

**Table 7. State Space of Scenario 1**

| markings | domains | classes | transitions |
|---|---|---|---|
| 399 | 455880 | 456926 | 978970 |

**Scenario 2** Scenario 2 includes the following initial values of parameters:

- SimulatedTrainSpeed = 0
- V_mrsp = 160
- SBAvailable = true
- DriveCommand = 1. The initial drive command is acceleration.
- T1 transition in Driver model has an effect behavior DriveCommand = 2. The time duration for initial behavior (acceleration) is 200000.
- T2 transition in Driver model has an effect behavior DriveCommand = 0. The time duration for the behavior before keeping speed (deceleration) is 100000.

The size of the state graph for scenario 2 is shown in the table below. The generation of the whole state space takes less than 26s (system time of 25.912s).

**Table 8. State Space of Scenario 2**

| markings | domains | classes | transitions |
|---|---|---|---|
| 474 | 700129 | 700472 | 1201679 |

**Verification of Requirements** We have used our model-checking toolbox to check the properties stated in the work by Univ. Bremen [7]. These properties are a direct translation into temporal logic of the requirements found on the Subset-026. Since the CSM model does not take into account the possible activation and de-activation of the CSM, we have not dealt with three requirements. (By default, the CSM is always activated.) We provide an interpretation of the nine remaining requirements using LTL, see Table 9. To simplify the the LTL formula, we have used simple names for naming the relevant variables. Full names, integrating information on the hierarchy, should be used when model-checking the actual systems in Tina.

## 3.5 Summary

What we have done:

1. Extended an environment model and a driver model in SysML for the CSM function.
2. Transformed SysML models to TPN models.
3. Verified LTL properties using Tina model-checking toolset on the TPN models.

## 3.6 Conclusions/Lessons learned

To provide deterministic verification scenarios, we have extended the CSM models defined by Uni Bermen with an environment model and a driver model. The transformation from semi-formal SysML model (including block, activity and state machine diagrams) to the TPN model is automatic. The resulting TPN models are then used to validate system specification written by LTL.

For now, there is still no state space explosion problem in this case study. We intend to verify the Speed and Distance Monitoring function model to further evaluate our method.

**Table 9. LTL**

| Requirement | LTL Formula |
|---|---|
| req_01 | EmergencyBrakeCommand ∧ SBAvailable=0 ∧ SimulatedTrainSpeed <= V_mrsp ∧ RevocationEmergencyBrake=0 |
| req_02 | ServiceBrakeCommand ∧ SBAvailable=0 ∧ SimulatedTrainSpeed <= V_mrsp |
| req_03 | ServiceBrakeCommand ∧ SBAvailable=0 ∧ SimulatedTrainSpeed gt (V_mrsp + dV_sbi) ∧ SimulatedTrainSpeed lt (V_mrsp + dV_ebi) |
| req_08 | ⋄ ((NORMAL ∧ SimulatedTrainSpeed <= V_mrsp) U (NORMAL ∧ SimulatedTrainSpeed gt V_mrsp + dV_warning ∧ SimulatedTrainSpeed <= V_mrsp + dV_sbi)) |
| req_09 | ⋄ ((NORMAL ∧ SimulatedTrainSpeed <= V_mrsp) ∧ ((NORMAL ∧ SimulatedTrainSpeed <= V_mrsp) U (NORMAL ∧ SimulatedTrainSpeed gt V_mrsp + dV_sbi ∧ SimulatedTrainSpeed <= V_mrsp + dV_ebi))) |
| req_10 | ⋄ ((NORMAL ∧ SimulatedTrainSpeed <= V_mrsp) ∧ ((NORMAL ∧ SimulatedTrainSpeed <= V_mrsp) U (NORMAL ∧ SimulatedTrainSpeed gt V_mrsp + dV_sbi))) |
| req_11 | ⋄ ((OVERSPEED ∧ SimulatedTrainSpeed <= V_mrsp) ∧ ((OVERSPEED ∧ SimulatedTrainSpeed <= V_mrsp) U (NORMAL ∧ SimulatedTrainSpeed gt V_mrsp + dV_sbi ∧ SimulatedTrainSpeed <= V_mrsp + dV_ebi))) |
| req_12 | ⋄ ((OVERSPEED ∧ SimulatedTrainSpeed <= V_mrsp) ∧ ((OVERSPEED ∧ SimulatedTrainSpeed <= V_mrsp) U (NORMAL ∧ SimulatedTrainSpeed gt V_mrsp + dV_sbi))) |
| req_13 | ⋄ ((WARNING ∧ SimulatedTrainSpeed <= V_mrsp) ∧ ((WARNING ∧ SimulatedTrainSpeed <= V_mrsp) U (WARNING ∧ SimulatedTrainSpeed gt V_mrsp + dV_ebi))) |

### 3.7 Future Activities

1. Refine the environment model by describing desired test scenarios using the test scenarios defined by Uni Bremen.
2. Model and verify the calculation of train position either by using the existing SCADE model or by refining the SysML model.
3. Encapsulate the transformation tool as an independent eclipse plugin.

# 4 Conclusion

## References

[1] UNISIG. SUBSET-026 – system requirements specification. SRS 3.3.0, ERA, March 2012.

[2] European union, commission decision of 25 january 2012 on the technical specification for interoperability relating to the control-command and signalling subsystems of the trans- european rail system - *2012/88/EU* , official journal of the european union, pp. l51/1-l51/65, 2012.

[3] European union, commission decision of 6 november 2012 amending decision 2012/88/eu on the technical specifications for interoperability relating to the controlcommand and signalling subsystems of the trans-european rail system - *2012/696/EU* , official journal of the european union, pp. l51/1-l51/65, 2012.

[4] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge, 2005.

[5] Hardi Hungar et. al. *openETCS Validation& Verification Plan*, version 00.09. `https://github.com/openETCS/validation/blob/master/VerificationAndValidationPlan/WP41-VerificationAndValidationPlan.pdf`, September 2013.

[6] Brice Gombault. *Safety analysis of Subset 026, Section 3.5, Management of Radio Communication (MoRC)*, version 4a.

[7] Cécile Braunstein, Jan Peleska, Uwe Schulze, Felix Hübner, Wen ling Huang, Anne E. Haxthausen, and Linh Vu Hong. A SysML test model and test suite for the ETCS ceiling speed monitor. Technical report, Univ. Bremen, 2014.

[8] Ning Ge. *Property Driven Verification Framework: Application to Real-Time Property for UML-MARTE Software Designs*. Ph.d., Institut National Polytechnique de Toulouse, Toulouse, 2014.