OETCS/WP4/D4.3.1

openETCS

ITEA2
INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

ITEA2 Project
Call 6 11025
2012 – 2015

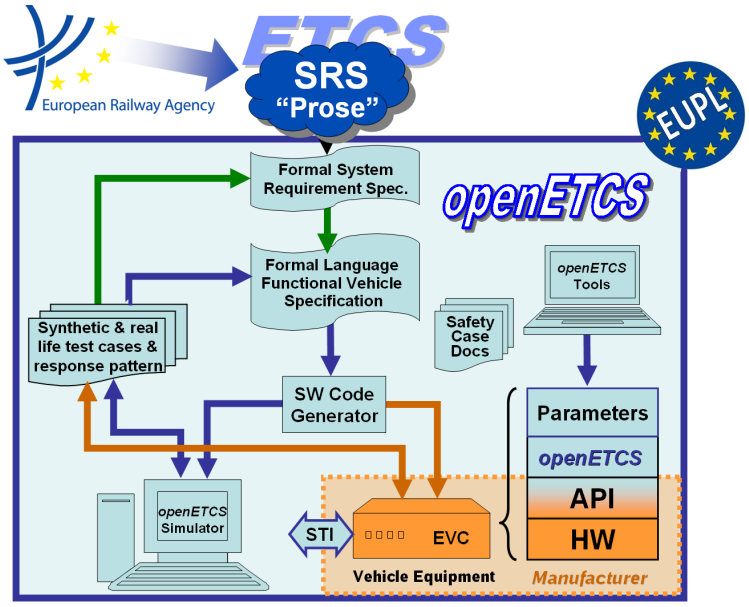
openETCS@ITEA Work Package 4.2: “Verification & Validation of the Formal Model”

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 ()	signature ()	signature ()	signature Klaus-Rüdiger Hase (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 2 — Institut Mines-Télécom: Verification of the Movement Authority (Subset-026 chapter 3.8)
- Section 3 – TWT GmbH Science & Innovation: Verification of Procedures (Subset-026 chapter 5)
- Section 4 – University of Bremen: Verification of the Management of the Radio Comumunication (Subset-026 chapter 3.5)
- Section 5 – University of Rostock: Verification of the Speed and distance Monitoring (Subset-026 chapter 3.13)
- Section 6 – Systerel: Verification of Procedure on-Sight (Subset-026 chapter 5), the Management of Radio Communication (Subset-026 chapter 3.5) and the management of modes and levels function (Subset-26 chapter 4.6 and several sections of chapter 5)
- Section 7 –LAAS and INPT: Verification of the Speed and Distance Monitoring (Subset-026 chapter 3.13)

Table of Contents

Figures and Tables

Figures

Tables

1 Introduction

This verification evaluated in five different approaches that the design conforms to an excerpt of Subset-026 [**?**] related to the specification [**?**] [**?**]. 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 Institut Mines-Télécom: Verification of the Movement Authority

2.1 Introduction

We focus on applying model-based formal methods on validation and verification (V&V) of the ETCS system. An overview of our approach is depicted in Figure 1.

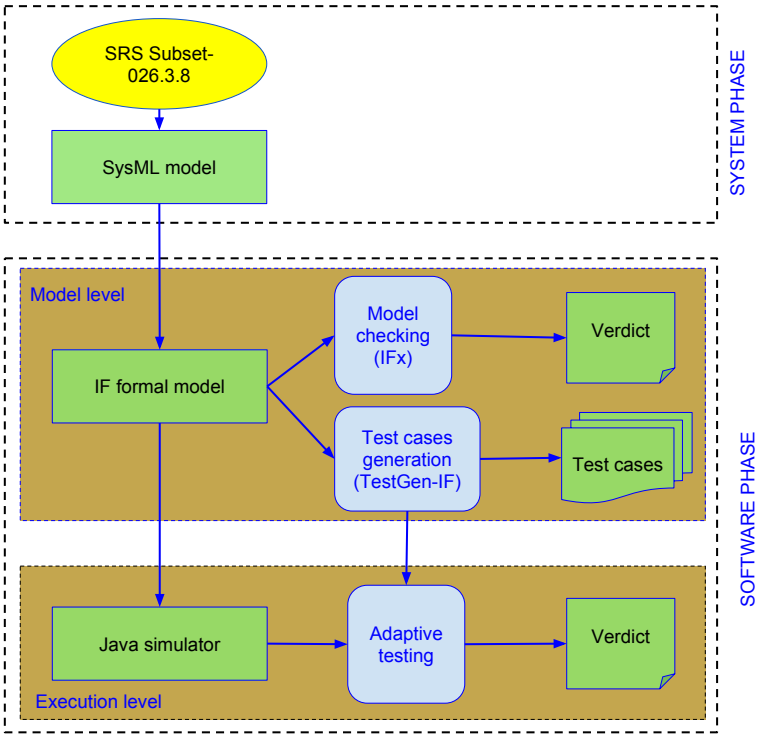


Figure 1. Overview of Institut Telecom-Mines' Approach

In the openETCS project, the system requirement specifications are represented by using SysML models. We validate and verify the models on two aspects, model-level and execution-level, by using two model-based techniques: model-checking and model-based testing. Model checking is a technique for automatically verifying whether a model of a system meets some given safety requirement. To solve such a problem, both the model of the system and the requirements are firstly formulated in some precise mathematical language. Model checking algorithms are then used to check the model against the requirement. Basically, all states of the model will be visited exhaustively to meeting the requirement or finding a counterexample. Model-based testing is a approach to test whether a system satisfies some requirements. It executes some test cases delivered from the requirements on the system under test to give verdict.

At the model-level, V&V is done through model-checking, by using IFx tool, of IF formal models which are representations of the SysML models. At the execution-level, we encode the SysML models by Java simulators that are then used to execute some tests. We also illustrate the consistency of two aspects by applying test cases, that are generated by TestGen-IF tool at the model-level, on our Java simulators, i.e., all tests must give *pass* verdict

The automatic translations from SysML models to IF models to Java simulators are being studied. Furthermore, as the actual models provided by WP3 have not yet been initiated, we started with a formal model that is a finite state machine augmented with continuous variables and guards. This model can be considered as an abstract version of ETCS model and it can be refined in our

future steps, e.g., the MOVE function mode of the TRAIN can be refined to SHUNTING, TRIP function modes of OBU in Subset-026 chapter 4.4.

2.2 Verification on IF model

2.2.1 Object of Verification

The object of verification is an IF model representing the Movement Authority of the train.

2.2.2 Available Specification

The Movement Authority is described in Subset-026 chapter 3.8. This chapter gives (1) the definitions and structure of the movement authority, (2) the procedures of OBU to send a Movement Authority request to the RBC and to receive a Movement Authority response from the RBC; and (3) the use of Movement Authority on the OBU.

2.2.3 Detailed Verification Plan

Goal

We intend firstly to model the Movement Authority of OBU described in Subset-026 chapter 3.8. We then use the model to validate the safety properties and to generate the test.

Means

We consider an ETFSM (Extended Timed Finite State Machine) as a formal model from which test cases for verifying the safety aspects of the developed implementations can be automatically generated. Formally, an ETFSM is a tuple $(S, s_0, E, T, \Delta, v_0)$ where: S is a non-empty finite set of states with $s_0 \in S$ as the initial state, E is a finite set of events, T is a set of transitions, $\Delta \subset S \rightarrow S \times (\mathbb{N} \cup \{\infty\})$ is timeout function, and $\vec{v}_0$ is the vector of initial values of the context variables. The timeout function Δ limits an interval by which a trigger of transition must occur (thus the transition will be fired). When the interval ends, the transition will be automatically fired in spite of its trigger has not yet occurred. The figure 2 represents the ETFSM model of the Movement Authority of OBU .

Method

Model checking is an automatic technique for verifying finite-state reactive systems. Using such techniques one could automatically check if the model specifies most of the requirements of the system, such as the important safety properties described in Task 4.4. Similar to proof techniques, in order to use model checking to verify if the SFM (Semi-formal model) and FFM (fully formal model) comply with the safety and function requirements, the properties to be proven have to be identified and described. To implement the use model checking, it is mandatory to specify the model using finite-state reactive systems, and they should also provide an intuitive way to express the properties to be model checked. The language for describing a formal model for which corresponding model checkers exist should be selected and the set of critical requirements to be verified need to be clearly identified. The proposed model checking techniques should be supported in the selected tool chain.

Tool

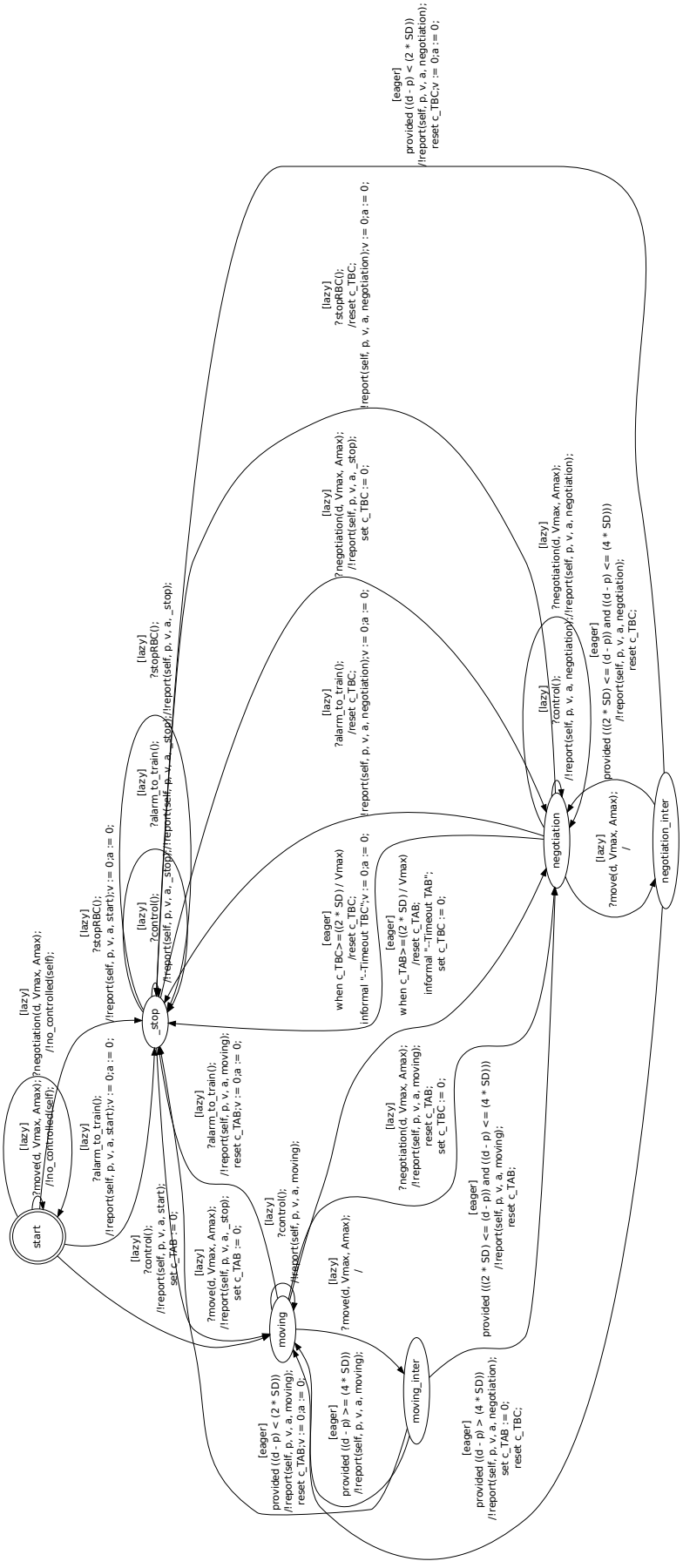


Figure 2. Extended Timed Finite State Machine of OBU

We use IF mdoel-checker () for model-checking and TestGen-IF for generating test cases. The formal model is hence represented by IF descriptionin order to be checked by using IF model-checker.

2.2.4 Results

We have provided a method to formally model the movement authority requirements of the train using finite state machines augmented with continuous variables (train position, speed, acceleration) and time constraints.

We have modeled OBU on Timed Extended Finite State Machine. We have then used IF model checker to verify the proposed formal model.

During the modeling we discovered 3 inconsistencies, ambiguities and gaps in the specification which we reported in [**?**].

Currently, the obtained model can be considered as an abstract representation of the system specification provided by the standard, i.e., the MOVE function mode of the OBU can be refined to SHUNTING, TRIP function modes of OBU in Subset-026 chapter 4.4.

2.2.5 Next Steps

On the one hand, we refine our formal model to take into account different function modes of OBU. In the other hand, we complete the automatic translations from the SysML specification to the IF specification or to our Java simulator.

2.3 Verification on Java simulator

2.3.1 Object of Verification

Model of the Movement Authority converted from IF language to the simulator.

2.3.2 Available Specification

Conversion of the IF model described on the previous section.

2.3.3 Detailed Verification Plan

Goal

We intend firstly to model the Movement Authority of OBU described in Subset-026 chapter 3.8. We then use the model to validate the safety properties and to generate the test.

Means

When using model checkers the criteria for the model to be safe is that all the safety properties should be checked. This is impossible, since the number of safety properties could be infinite and thus, only some of them can be checked through the above step. For this reason, as a complementary approach, testing is commonly used. If the corresponding model respects some requirements, i.e., only expected outputs are produced to applied input sequences, to some extent, there is a confidence that the models is safe. In order to derive ‘good’ tests a formal model

should be involved. It is known that only the FSM model where each input is followed by a corresponding output allows automatic deriving finite tests with the guaranteed fault coverage where the races between inputs and outputs can be easily avoided. Many authors for deriving finite tests with the guaranteed fault coverage turn their model to some kind of an FSM (see, e.g., [**? ? ?**]).

We decided to use TestGen-IF to automatically extract a set of test cases from the formal model described in the IF language. We have identified a set of basic requirements and we can describe them as IF properties. Based on these properties, the TestGen-IF tool automatically generates a set of test cases. TestGen-IF implements Hit-or-Jump [**?**] algorithm stategry to generate the test. These test cases can be used to test if the model follows the requirements defined for the test generation.

For simulation, the artifacts should provide means to execute the model. The simulator must be automatically generated, so that, when run against test scenarios (inputs/outputs for the model), we may conclude whether the model follows the specification or not. In particular, it is important to define test scenarios for the safety critical properties. The simulation should cover all states, transitions, data-flow, and paths in the model. It would also be desirable to include graphical representation of the simulation/model and also provide a report of the visited components. Being able to trace the requirements that are met during a simulation is also advisable to allow simple requirement coverage.

Once we have a test suite generated by TestGen-IF, we execute them using our simulator. The simulator provides a trace of the execution of each test and the expected trace. By comparing both traces it is possible to identify problems with the model.

2.3.4 Results

We have provided a method to formally model the requirements of the ETCS using finite state machines augmented with continuous variables (train position, speed, acceleration) and time constraints.

We have modeled OBU on TEFSM. With TestGen-IF we automatically generated a test suite that was used to verify our simulator.

By providing TestGen-IF with test objectives, we were able to automatically generate a test suite capable of verifying the properties related to them. Currently we have four different test objectives and with them we generated a suite of 22 tests. By providing more test objectives it is possible to generate more tests. Each test constitutes a sequence of inputs (or period of time without an input) and expected outputs.

The tests generated were executed by the simulator. When executing a test, the simulator provides a file that compares the expected trace generated from the model with the trace that was generated by the simulator. If an inconsistency occurs, the test is considered failed. On our first execution we found some inconsistencies between the IF model and the model used by the simulator. After taking care of these inconsistencies we were able to execute all the tests pass.

It is possible to find inconsistencies in a model using the Java simulator to execute tests. However, further testing is needed to determine the completeness of our test suite.

2.3.5 Next Steps

We plan to verify the fault coverage of these tests by executing Java simulator against corresponding traces and Java Mutants. For this stage we used an older version of the model. A newer version is currently being updated and will be used on future activities.

3 TWT GmbH Science & Innovation: Verification of Procedures of Subset 026 chapter 5

This sections reports on the modeling of the procedures described in Subset 026 chapter 5—that is, the behavioral part of the ETCS. The goal of the activity is to validate the specification and to support the modeling using SCADE and the verification of SCADE models on a higher level of abstraction in comparison to SCADE models.

The activity is described in the Verification and Validation Plan (see Sect. 6.1.2.5). In short, we provide feedback regarding ambiguities, inconsistencies and errors in the current ETCS standard based on our formalization of the specification using mathematical modeling languages.

3.1 Object of verification

The object of verification is Subset-026 chapter 5 of the specification. We formally model parts of the specification and use the resulting model to validate the specification. This design step has been described in D2.3 (see Sect. 4.4).

3.2 Available specification

The specification is described in Subset-026 chapter 5. It describes procedures of ETCS entities (i.e., required reactions on events and received messages), thereby focusing on required change in status and mode of entities considered.

3.3 Detailed verification plan

3.3.1 Goals

The goal is to model the the procedures described in Subset-026 chapter 5, thereby focusing on modeling the *system behavior*—that is, the control flow of the on-board unit and the interplay with its environment (e.g., the driver and the RBC). The model is then used to validate the specification.

3.3.2 Method/Approach

As a formal model, we use *colored Petri nets* (CPNs) [**?**], an extension of classical Petri nets [**?**] with data, time, and hierarchy. CPNs are well-established and have been proven successful in numerous industrial projects. They have a formal semantics and with CPN Tools [**?**], there exists an open source tool for modeling CPNs. Moreover, CPN Tools also comes with a simulation tool and a model checker, thereby enabling formal analysis of CPN models.

We focus on modeling the *system behavior*—that is, the control flow of the on-board unit and the interplay with its environment (e.g., the driver and the RBC). Figure 3 depicts the CPN representing the highest level of abstraction. It shows the decomposition of the overall system into the on-board unit and its environment: the driver, the RBC, the RIU, the STM, and the GSM module. Each component is modeled as a subpage (i.e., a component). Graphically, a subpage is depicted as a rectangle with a double-lined frame. Furthermore, the model shows through which message channels and shared variables the on-board unit is connected to its environment. A channel or shared variable is modeled by a place which is graphically represented as an elipse. As an example, the driver (i.e., subpage `Driver`) may send a message to the on-board unit (i.e.,

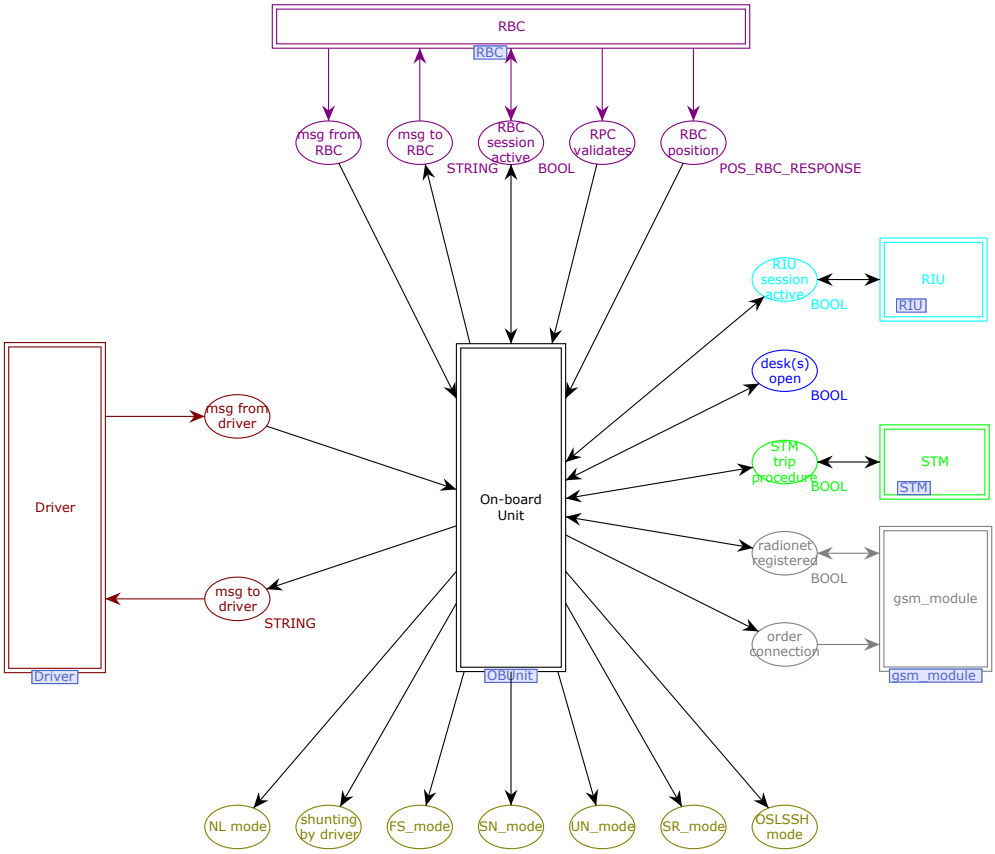


Figure 3. Top level model

subpage `On-board Unit`) via the place `msg from driver`, and receives messages sent by the on-board unit via the place `msg to driver`.

Zooming in subpage `On-board Unit` yields the CPN model in Fig. 4. This CPN model has two subpages: Subpage `Start` models the states S0 and S1 of the specification (i.e., Subset-026 chapter 5.4) and subpage `Rest` the remaining states. At this level of abstraction, we see on the left hand side seven places (green frame). Each such place models (a part) of the state of the on-board unit, for example, the mode and the train running number. The current model has 689 places, 173 transitions and 1,227 arcs.

Having a more detailed look at Fig. 4, we observe that our model does not represent all variables of the on-board unit as given in the specification and also partially abstracts from data. We abstract from those details, because the model is tailored to formalize the *control flow* of the on-board unit and, in particular, the *communication behavior* with its environment. As a benefit, this abstraction reduces the complexity of the model and improves its understandability. Additional details, such as data and precise message values, can be added in a refinement step.

The modeled procedures have been manually modeled using CPN Tools. Thereby, each element in the model has been reviewed against the respective requirement, as given in the specification. To improve the confidence in the model, in a second step, a person other than the modeler checked the model against the specification. In addition, we used the simulator to check whether the modeled behavior of the CPN matched the intended behavior.

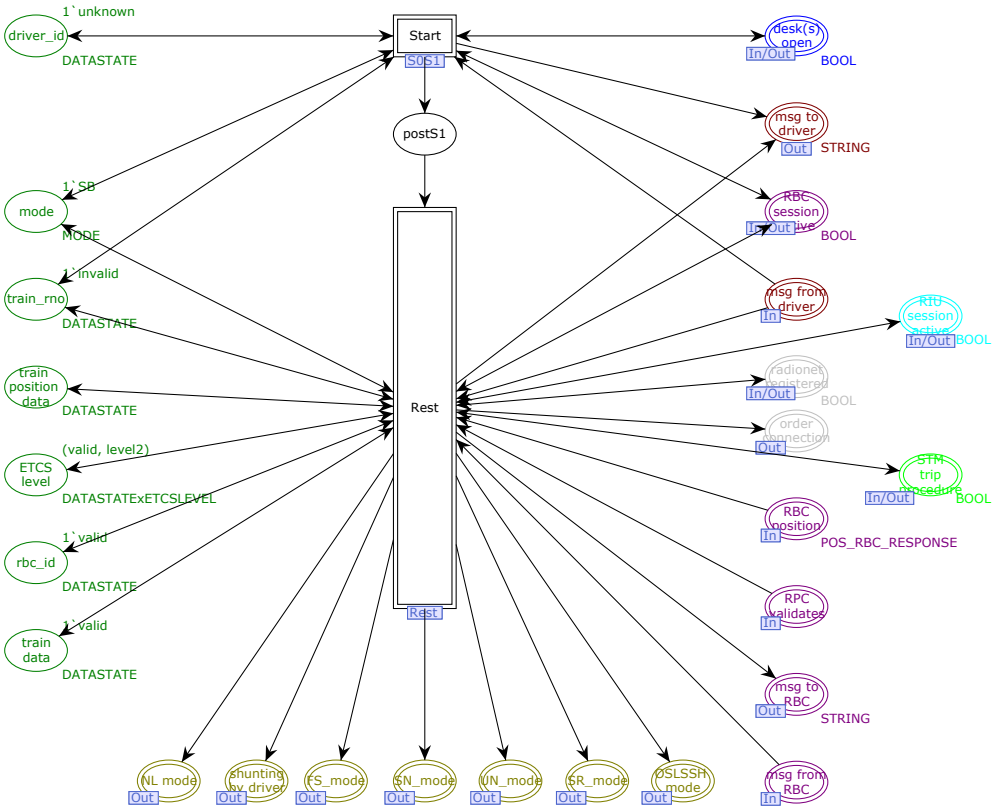


Figure 4. CPN model of the on-board unit

So far, the primary goal of modeling has been to validate the specification. During the modeling we discovered 36 inconsistencies, ambiguities and gaps in the specification which we reported in [**?**].

3.3.3 Means

The input for our approach is the specification described in Subset-026 chapter 5. Our output is a CPN model and a report describing inconsistencies, ambiguities and gaps in the Subset-026 chapter 5.

3.4 Results

3.4.1 Summary

We have modeled the following five procedures of Subset-026 chapter 5 as CPN:

- Start of Mission (Subset-026 chapter 5.4)
- End of Mission (Subset-026 chapter 5.5)
- Shunting Initiated by Driver (Subset-026 chapter 5.6)
- Override (Subset-026 chapter 5.8)
- Train Trip (Subset-026 chapter 5.11)

Our CPN models the system behavior—that is, the interplay between the different entities of the ETCS—and partially abstract from data. Therefore, we complement the work on SysML modeling, where the focus is on the connectivity of components and the data types.

3.4.2 Conclusions/Lessons learned

The numerous specification findings illustrate the need for validating the specification. CPNs are well-suited to model the behavioral aspects described in Subset-026 chapter 5. The size of the model clearly indicates the complexity of the procedures, even at the current level of abstraction. Therefore, we expect that applying formal verification on the resulting CPNs will not be feasible due to state-space explosion.

3.5 Future Activities

We shall continue our work by completing the model, contributing to the modeling of (parts of) Subset-026 chapter 5 using SCADE, and verifying the SCADE model. In addition, we are planning to exploit synergies by collaborating with the project partner LAAS who advocate the Petri net model checker TINA [**?**]. In addition, we are working with the project partners from Braunschweig University of Technology on generating test cases from the CPN model.

3.6 Modeling the Subset-026 chapter 5

We plan to model the remaining parts of Subset-026 chapter 5, thereby reporting possible additional findings in the specification. The goal is to have a CPN modeling all procedures that are described in Subset-026 chapter 5. We also want to compare our model with the (corresponding part of the) ERTMSFormalSpec model [**?**].

3.7 SCADE Modeling

As the ETCS will be modeled using SCADE, we shall contribute to this modeling process. To use the experience that we gained from modeling Subset-026 chapter 05 with CPN Tools, we want to contribute to the SCADE modeling of (parts of) the Subset-026 chapter 5. The SCADE design flow starts with modeling all components and their interplay using SysML block diagrams (with the tool SCADE Designer). The resulting SysML diagrams provide a functional and an architectural view. They are similar to the CPN model in Fig. 3. In a second step, the behavior of each block has to be fully modeled on the system level using SCADE Suite. Currently, SCADE does not support state machine models on the level of SysML. Our CPN model provides this level of abstraction and will, therefore, be useful for the SCADE modeling.

3.7.1 Verification of the SCADE Model

Another task concerns the verification of the resulting SCADE model. Recently, researchers reported on complexity problems already for medium-sized SCADE models that restrict the verification using the SCADE prover [**? ?**]. Given the complexity of the ETCS, we assume that we will face similar challenges. To alleviate those complexity problems, we aim to apply the following three techniques:

Abstraction: We will apply abstraction techniques on the SCADE model to prove safety properties on a higher level of abstraction whenever possible. On the one hand, we can apply SCADE contracts to restrict the domain of the input values. This technique is known as environment abstraction. On the other hand, we can transform the SCADE model into a

model of higher abstraction, thereby using different formalisms such as timed automata, transition systems and Petri nets. (As SCADE has a formal semantics such transformations are possible, but may take considerable effort.) We can then use verification tools that are dedicated to the properties of interest and the chosen formalism. We see the chance that our CPN model can be used for this task, too. For example, Uppaal [**?**] can analyze timed automata, the Spin model checker [**?**] and NuSMV [**?**] can analyze transition systems, and TINA [**?**], LoLA [**?**], and CPN Tools [**?**] are tools for analyzing (different variants of) Petri nets.

Compositional Reasoning: Another approach is to prove properties for individual components and deduce from it the correctness of a property concerning the entire ETCS. Here we think that we can, in particular, combine our model with the MoRC model [**?**] and apply compositional reasoning.

Correctness by Design: The two previous approaches support *correctness by verification*; that is, first the model is designed and in the next step it is verified. A different methodology is *correctness by design*. The idea is to model on a higher level of abstraction and to prove that certain safety properties hold. Then the model is iteratively refined. Each refinement step has to guarantee that all properties that hold for the more abstract level also hold in the refined model. The challenge is to find property-preserving refinement rules or a refinement relation between an abstract model and a refined model that preserves the desired properties and to verify that this relation holds. The results can be applied to validate the SCADE model and the specification.

4 University of Bremen: Verification of the Management of the Radio Communication

This section reports the verification activity of SCADE-MoRC. The goal of this activity is, first, to establish the compliance of the SCADE model to the SRS description via testing. Secondly, we want to track the ambiguities within the specification. Finally we want to demonstrate the efficiency of model based testing using the RT-tester tool for system integration testing.

The activity is described in the Verification and Validation Plan section *6.1.2.7 System Integration Testing (Uni Bremen/DLR)* [**?**]. In short, we develop a test model from the specification, generate tests and use the code generated from SCADE to perform software-in-loop testing. The test model and the SCADE model used to generate code have been done independently to each other.

4.1 Object of verification

Management of radio communication (MoRC) ERTMS function baseline 3.

The system under test (e.g. the verification object) is the C code generated from a SCADE model and described here `https://github.com/openETCS/model-evaluation/tree/master/model/SCADE_Siemens/Subset_026_Chapt_3.5_ManagementOfRadioCommunication/Generated_C_Code`. It describes the Management of the radio communication at the software phase.

4.2 Available specification

The specification is described in the Subset-026 chapter 3.5. It describes the communication protocol between the EVC and the RBC or balises. In particular, how the EVC initiates and terminates a communication.

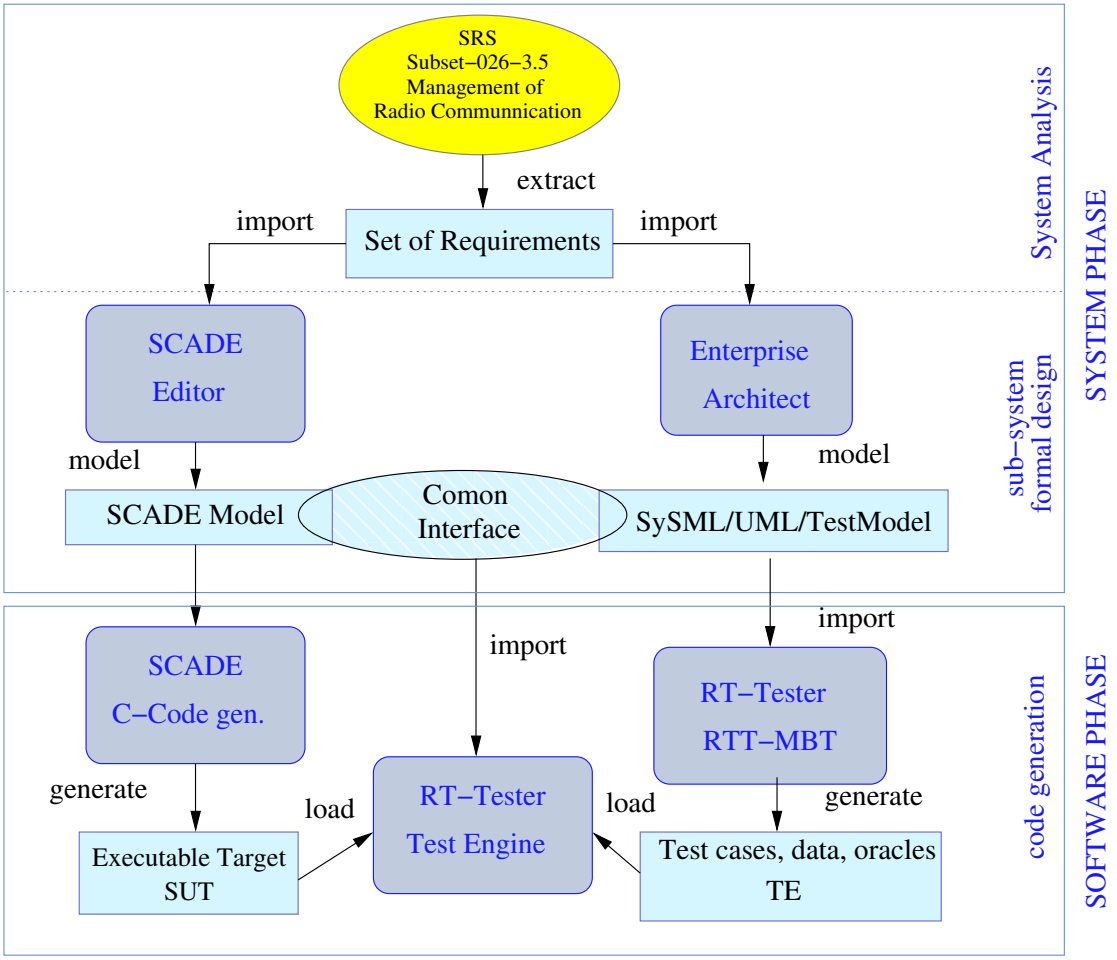


Figure 5. SCADE/RT-Tester methodology

4.3 Detailed verification plan

4.3.1 Goals

To achieve what has been defined previously a test model in SysML has been developed. The description of this verification artifact may be found here `https://github.com/openETCS/model-evaluation/blob/master/model/EA-SysML/new_version/doc/ea_sysml_report.pdf`

Our main goal is to verify the SCADE model by test simulation. The tests are produced by a model of the Subset-026 chapter 3.5 described as a state machine.

4.3.2 Method/Approach

Figure 5 depicts our methodology. From the SRS specification, two models are designed: one SCADE model that will be then used to produce C code and one SysML model for generating tests.

Our test model contains the behavioral representation of the MoRC, the set of requirements of the chapter 3.5, and the link between the behavior and the requirement. Most of the requirements may be represented as single transition or state in the test model state machine. Nevertheless, some of the requirements may be only modeled as a path of the state machine, we choose to represent

those as LTL formula, added as constraints of the test model. For example the REQ-3.5.3.10 describes the steps needed for the establishment of the radio communication order by the RBC. This requirement explicits a particular path of the test model, thus this path should exists in our test model. To ensure that this particular test will be generated a LTL formula has been added (See [**?**] for more details).

REQ-3.5.3.10 "If the establishment of a communication session is initiated by the RBC, it shall be performed according to the following steps ..."

```
Finally (MessageIn == INIT_SESSION_TRACK && setUp == 1 ->
Next (MessageOut == SESSION_ESTABLISHED && radioComSession == ESTABLISHED)
```

We need then to link the system under test and the test model. Since the two models are elaborated independently, we need to ensure that the tests generated may be handled by the code generated by SCADE and conversely we need to read the output of the C code back to the test generator to compare the values with our oracles. This is achieve by defining a common interface that the two models should respect. The inputs drive the tests and the outputs are the observational points to state if a test pass or fail. Hence, the two models should respect the same interface.

After the modeling phase, starts the test generation phase. Two strategies have been used for the tests generation. First, we generate a set of test following the common behavioral coverage strategies ensuring the following coverage :

- Basic control state coverage (BCS): All state locations are covered.
- Transition coverage (TC): All transition of the statecharts are covered.
- Modified condition/decision coverage (MC/DC): Modified condition/decision coverage.
- Hierarchic transition coverage (HITR): High-level transition of nested statecharts are covered.
- Basic Control states Pair coverage(BCPAIRS): For concurrent state machines pair states of two different state machines.

More detailled explaination on the coverage may be found here [**?**].

We then apply a requirement coverage strategy to generate tests that covers all the requirements. As each requirement is linked to an artifact of the test model, part of the test cases are generated as subset of the coverage mention above. In addition, test cases from the LTL are also provided.

Finally the test engines run the SUT with the stimuli from the test model. In parallel, it runs the test oracles that states if the test pass or fail.

4.3.3 Means

The Artifacts are produced as follow:

- C code comes from SCADE model.
- Test model is a SysML model designed with Enterprise Architect.

- Test cases, tests data and test oracles are generated with RT-tester.
- Executable code compile with gcc.
- Code run within the RT-Tester engine.

SCADE is EN 50128 qualified at SIL 3/4, RT-tester is also certifiable T3 as shown in [**?**].

4.4 Results

Coverage strategies	# tests generated
BCS	14
TC	40
MC/DC	79
BCPAIRS	33
HITR	12
LTL	2
Total Tests	180

Table 1. Test cases generation summary

Table 1 resumes the set of automatically generated tests. The set of tests cover 40 of the requirements from the chapter 3.5.

The simulation result of the C code is not yet finished and for the moment, all test failed. The main reason was that the two models did not share the same starting condition. Hence, we need to refined our test model to be able to handle SCADE modeling style correctly and be able to have interesting result.

4.5 Summary

What we have done:

1. Created a test model in SysML.
2. Generated test cases.
3. Ran SCADE model against test procedure produces by RT-tester.

The next step:

1. Refined test model
2. Analyze the result of the test procedure.
3. Coordinate DLR/SIEMENS/Uni Bremen interfaces.
4. Run the test on the DLR lab.

4.6 Conclusions/Lessons learned

Our first attempt to simulate the tests was not a success. All tests except the one covering the initial states failed. Our two models have, at least, the same initial state. From our first investigation, we could see that one chapter of the specification is not self-contained. This leads to different interpretation in the modeling and thus some non equivalent behaviors.

Moreover some missing information in the specification leads to constraints in test model. It affects the test generation by providing some non realistic test cases. Some variable behavior that were not mentioned in the specification are considered as free and may have any possible value in their definition range. This can be solved by adding information into the test model.

More precisely, the test model is composed with a system under test and a test environment. In our first model the test environment is empty, meaning that all inputs of the SUT are free. The test enviroment may be described (and constrainted) by statecharts or LTL formula that restricts the behaviors of the inputs. The test generator should then find test suites that realize the given coverage and that respect the constraints given in the test environment.

4.7 Future Activities

4.7.1 Refine the test model

1. Analyze the test results of the SCADE C code
2. Enumerates specification ambiguitites: where the two parties did not undestand the specification in the same way.
3. Refine the test model by adding a better test environment with the help of domain experts

4.7.2 New activities

We will also provide the ceiling speed monitoring function to enrich the test model and apply new model based testing approach.

5 University of Rostock: Verification of the Speed and Distance Monitoring

This section reports the verification activities of the *Speed and Distance Monitoring* with model based simulation and virtual prototyping. The first activity pursues the goal of formalizing the specification in the form of an executable model. This model provides a performance estimation at an early stage of system level design and adduces evidence what hardware resources (hardware platform) will be needed for the future OBU. Secondly, finding and reporting of unclarities, inconsistencies, incompleteness and errors while implementing the specification by using tools of the openETCS toolchain (Papyrus/SysML, Scade Suite). Furthermore, we are developing a method of SystemC code generation from abstract and domain specific SysML models. Finally we want to demonstrate the efficiency of model based simulation after transformation from SysML to SystemC models.

The activity is described in the Verification and Validation Plan section *5.3.10 Verification with Model-Based Simulation* [**?**]. To sum up, we develop application models from the specification of the *Speed and Distance Monitoring*, generate test scenarios and use the inherent simulation environment of SystemC to do performance and scheduling analysis.

5.1 Object of verification

Speed and Distance Monitoring ERTMS function baseline 3. The system under test is the implemented SystemC code which is described here: `https://github.com/openETCS/model-evaluation/tree/master/model/SystemC_TWT_URO/3.13_Speed_and_distance_monitoring`. It describes the Speed and Distance Monitoring at the Software phase.

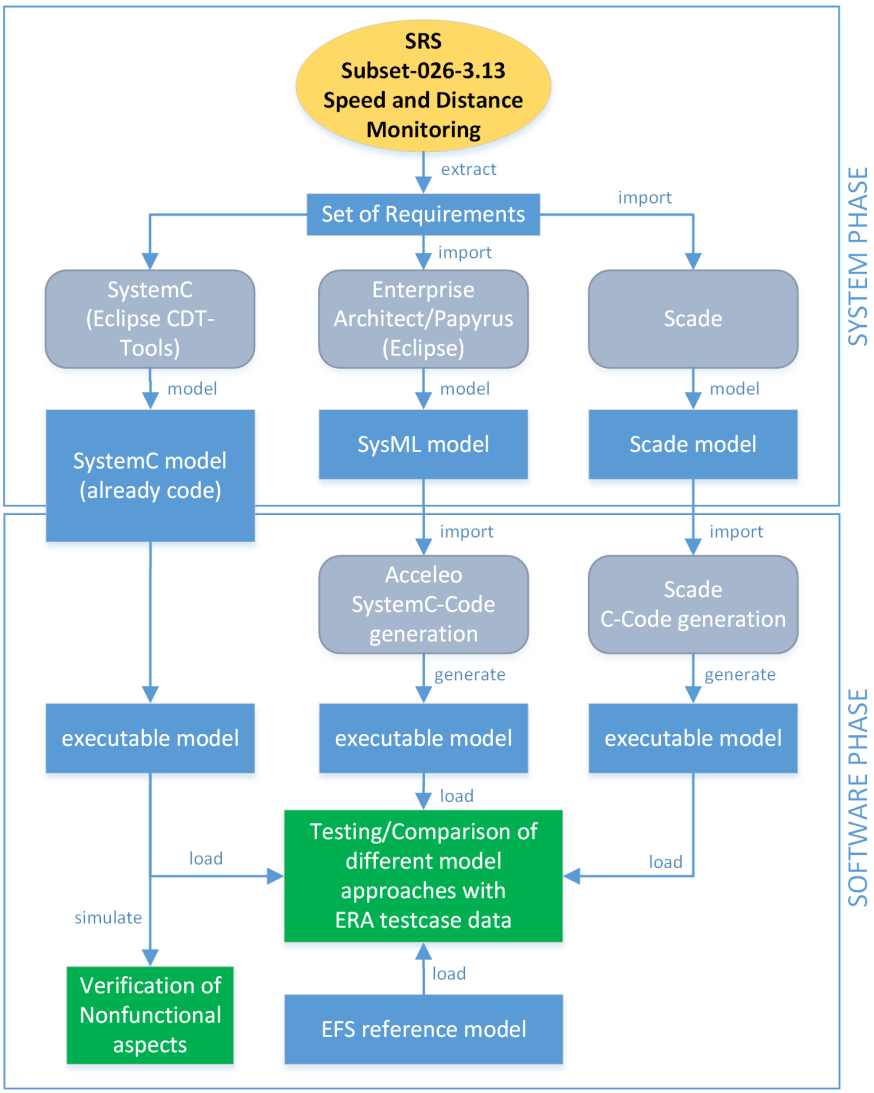


Figure 6. University of Rostock VnV Approach

5.2 Available specification

The specification is described in the SUBSET-026 Chapter 3.13 [**?**]. It describes the realization of both Train Interface (TI) and Driver Machine Interface (DMI) commands by calculating several modules with inputs form train side, track side and odometry. The University of Rostock focuses on the calculation of parts, which are used for safety critical cases using emergency brakes. This especially includes the EBD (emergency brake deceleration) curves.

5.3 Detailed verification plan

5.3.1 Goals

On the one hand we are testing extra-functional aspects such as performance and scheduling analyses to give evidence which hardware system is sufficient to meet the system requirements. We want to discover which hardware resources (e.g. number of processors) will be needed for the OBU. This is important to avoid excessive delays to ensure adequate response times in critical situations. Therefore the University of Rostock will do model based simulation using the inherent simulation environment of SystemC.

On the other hand we use the existing SystemC model to check against a reference model, such as EFS braking curve model. Furthermore we use different tools and means to build additional system models for comparison and verifying behavior.

5.3.2 Method/Approach

Figure 6 depicts our methodology. Three models will be created from the specification (SRS): one SystemC model that is directly implemented (hand written) into executable code (finished), one SysML model that will be used to produce (automatically) SystemC code to be executable (not finished) and one Scade Suite model that will be used to produce C code (not finished). For model verification we use test cases and data provided by WP4 and use the ERA excel datasheet as reference implementation. The created test models will contain behavioral representation of the *Speed and Distance Monitoring* such as state machines, activity diagrams and sequence diagrams. There will be a link to the specification requirements to meet the needs of traceability in terms of verification activities.

5.3.3 Means

The Artifacts are produced as follow:

- SystemC code which is directly implemented (hand written source code).
- One test model is the generated SystemC model. It's code will be generated/transformed by Acceleo from abstract SysML models designed with Enterprise Architect and/or Papyrus.
- C code is produced by Scade using Scade Suite.
- Executable code is compiled with GNU-C-Compiler `gcc`.
- Reference model (for now because no other is available) delivered by ERTMS Formal Specs.
- Testdata and testcases are provided in corporation with ERTMS Formal Specs using the ERA excel data sheet.

5.4 Results

Verification of extra-functional aspects is successfully done for the first iteration of application models. The provided results consist of recommendations on how hardware resources shall be allocated to the future OBU. The modeling activities are still in progress.

5.4.1 Summary

What we have done:

1. Created an executable model in SystemC.

2. Ran simulations on single, dual and multi core virtual prototypes.
3. Architecture SysML model of the *Speed and Distance Monitoring*.
4. Architecture Scade model of the *Speed and Distance Monitoring*.
5. Defined a reduced set of parameters for calculating braking Curves especially EBD curves.

The next step:

1. Finishing model activities on SysML and Scade.
2. Developing a model transformation/code generation from SysML models.
3. Defining an exchanges interfaces between different model approaches.
4. Run the tests.

5.4.2 Conclusions/Lessons learned

From the first results, we see that SysML is a very powerful graphical modeling language but to perform code generation it is necessary to have restrictions to it. We will have a domain specific SysML profile to get reliable results from code generation.

5.5 Future Activities

Simulink as a modeling tool is in our interest because there is a bridge to Scade. Simulink provides code generation for hardware description languages such as VHDL. That enables new hardware test scenarios.

6 Systerel

Railway and aerospace critical software require rigorous design, verification and validation processes. These can be achieved by the use of formal methods as recommended by the standards in these domains [**? ?**].

One possibility is to apply formal methods from the early stage of the design to produce “correct-by-construction” software. Numbers of success stories have already been described, for example in the railway industry with the B method [**? ? ?**]. Another approach is to introduce formal methods later, during an independent verification and validation phase: starting from an informal specification, the properties to verify are identified and formally specified in parallel to the software design activities. Then, they are automatically checked on an executable model or the code.

In the following we will describe the two approaches applied on three models:

1. VnV on classical B model that cover software design level, in the objective to provide an open-source approach; this model specifies the procedure "On-Sight" (Subset-026 chapter 5.9).
2. VnV on Event-B model that cover system level and support safety analysis; this model specifies the "Management of radio communication function" (Subset-26 chapter 3.5).

3. VnV on Scade model that cover software design level; this model specifies the "modes and level management" function (Subset-26 chapter 4.6 and several sections of chapter 5).

6.1 Verification and Validation on Classical B model

The first section 6.1.1 describes usual verification and validation activities applied during the design of industrial critical software using the B method. Second section 6.1.2 gives a description of tools available.

Last section 6.1.3 describes the results on an example.

6.1.1 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 [**?**]. 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.

6.1.2 Tools

In this section, we present the existing tools suitable for the verification processes defined in Section 6.1.1.

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.

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.

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

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 2 shows, for each of the verification processes presented in 6.1.1 (specification and source code were added as artifacts which support the activity), the corresponding item in the CENELEC norm annex table.

		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 2. Correspondence between CENELEC norm recommendations and the presented verification processes

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

[2]For additional information on qualification, see subsection 6.1.2 or the CENELEC norm.
[3]See `http://www.formalmind.com`

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 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 3 summarizes the presentation of the tools in subsections 6.1.2 and 6.1.2.
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 3. Comparison of the tools available for B verification processes

6.1.3 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.

Presentation of the B model As shown in figure 7, 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),

[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`.

- `os_train_info`: elaborates the location and the speed of the train.

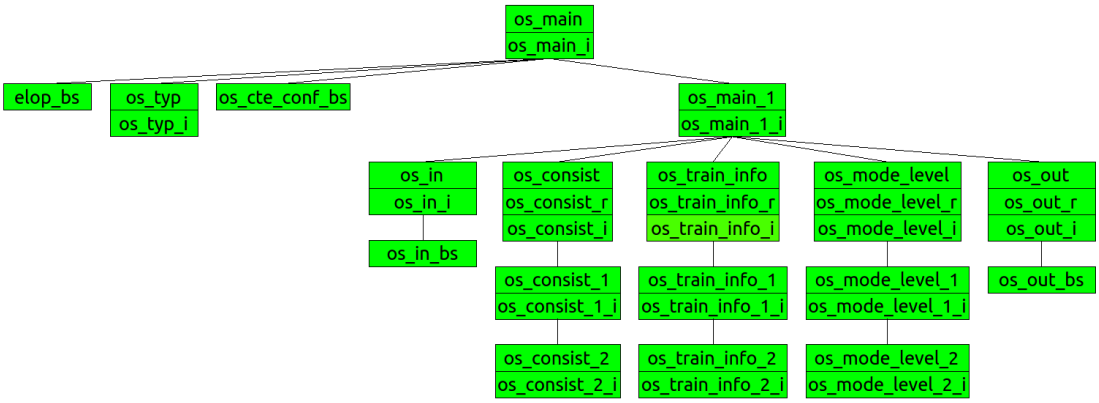


Figure 7. Architecture of the B model for the Procedure On-Sight example

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 8 shows the fully-proved model in Atelier B.

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`.

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

Below is presented a very simple theory (among several others) used for the proof of this component:

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

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

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

6.2 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.

6.2.1 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.

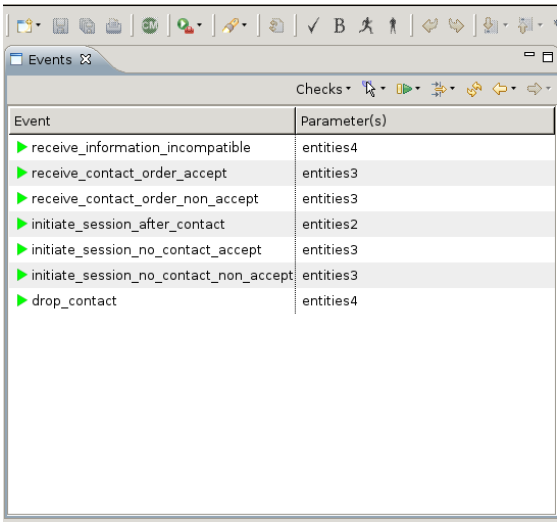


Figure 9. ProB Model Animation

Figure 9 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.

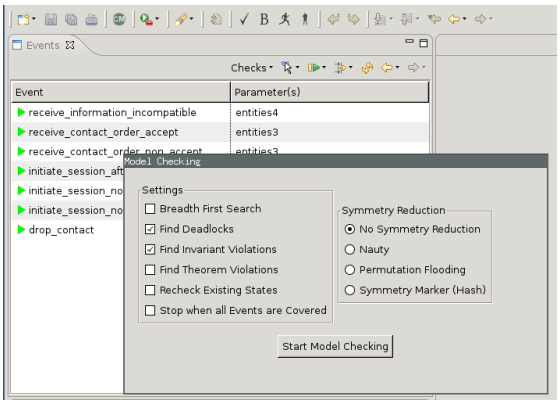


Figure 10. Model-Checking for Deadlocks

Figure 10 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.

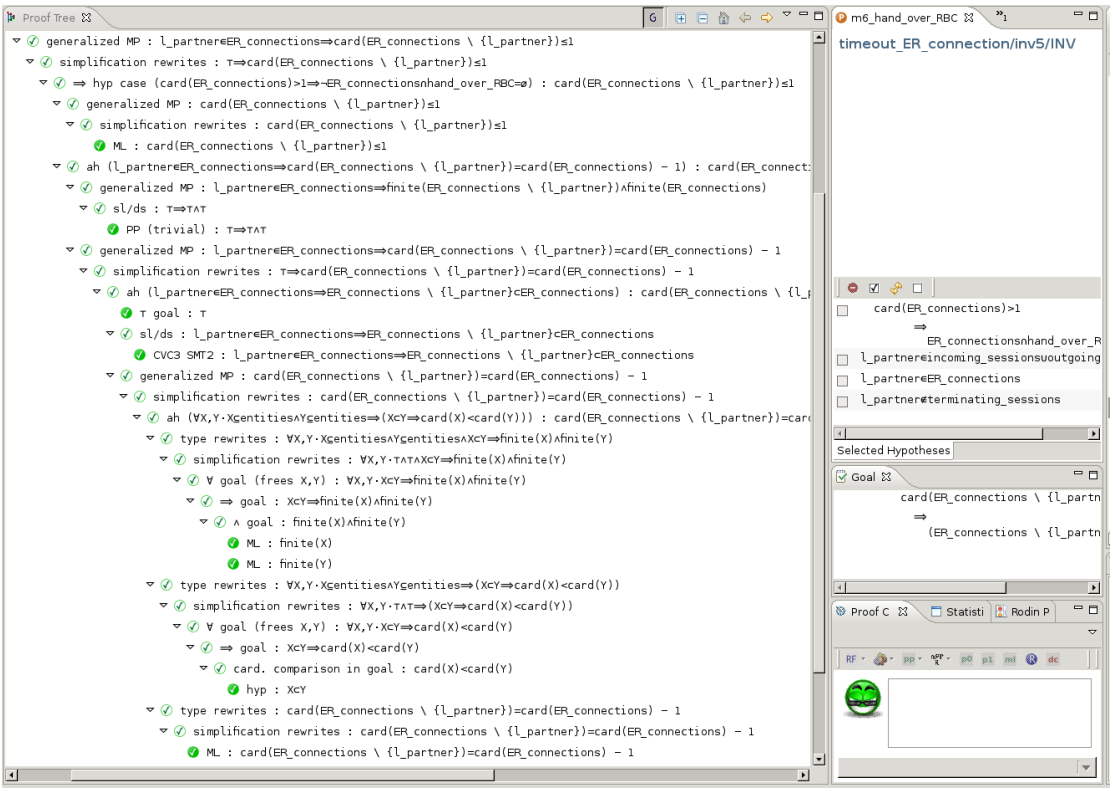


Figure 11. Rodin Proof Tree

Figure 11 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.

6.2.2 Data Refinement

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.

6.2.3 Code Refinement

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

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 12. Event Refinement

Figure 12 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.

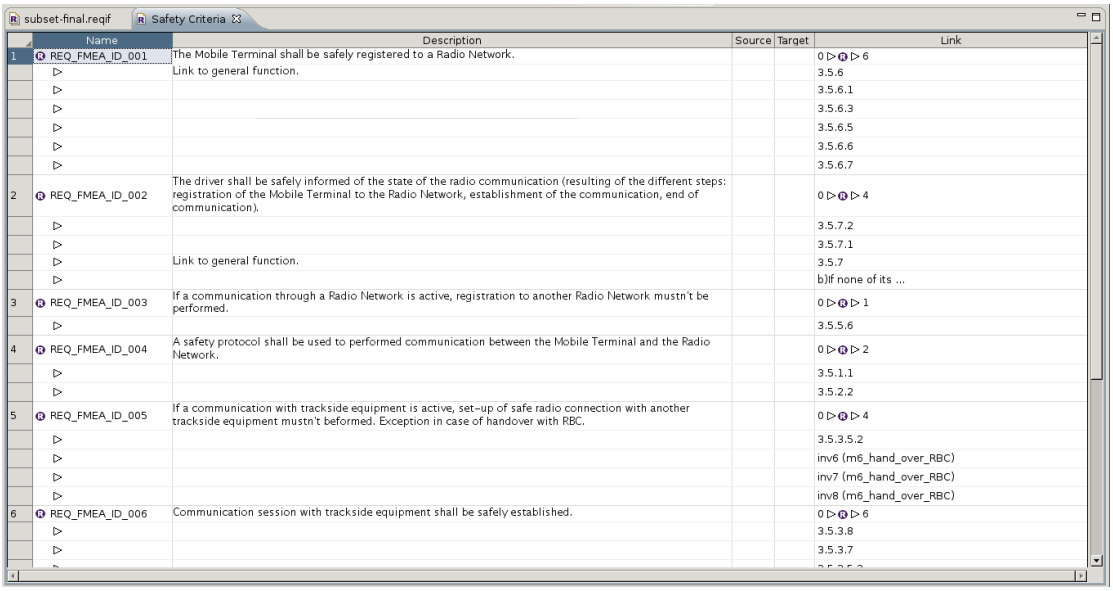
The activity is described in the Verification and Validation Plan [**?**]. In short, it consists of formalizing and proving the identified requirements of a preceding phase, to ensure their correct implementation of the requirements. To achieve this, we use the direct connection between the Event-B model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

Figure 13 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



	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 13. Safety Requirements

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 [**?**]. 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.

6.2.4 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`

6.2.5 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.

6.2.6 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.

6.3 Classical 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.

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.

6.3.1 Respect of modelling rules

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

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.

6.3.2 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.

6.3.3 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.

6.3.4 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.

6.3.5 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.

6.4 Formal verification processes applicable to a SCADE model

A second possibilities is to apply a formal verficiation process to the Scade model, for this we propose an a posteriori verification based on model-checking.

6.4.1 Principles of S3

Systerel Smart Solver (S3) is a SAT-based model checker for safety properties analysis.

The High Level Language (HLL), the input language of S3, is a stream oriented declarative data-flow language which can be used to model:

- the system behavior
- the environment
- the formal expression of the properties.

S3 proceeds to the analyses of properties on the traces of the HLL models following one of these two strategies:

- Induction to prove a property (see [**?**]);
- Bounded model checking (BMC) to falsify a property or generate test cases (see [**? ?**]).

Thus S3 can be applied on different ways to verify and validate a critical industrial system.

6.4.2 Application of S3: Static analysis

S3 adds some proof obligations to assess that the HLL model is correctly defined:

- Indexes of arrays belong to their ranges
- Latch definition range check
- No division by 0
- No overflow and no underflow on arithmetic expressions
- Output and constraint initialization check

Besides, the translators from a language to HLL can also generate proof obligations to be analyzed by S3, to check that the code does not have undefined behavior with respect to the source language.

For example the C-translator adds some proof obligations to ensure conformance with the C99 standard.

6.4.3 Application of S3: Verification of Safety Properties

Safety Properties are of the form *always* ϕ meaning that the Boolean predicate ϕ holds Globally, *i.e.*, in every state reachable from the initial state. S3 can be used to verify safety properties as shown in the process in Fig. 14.

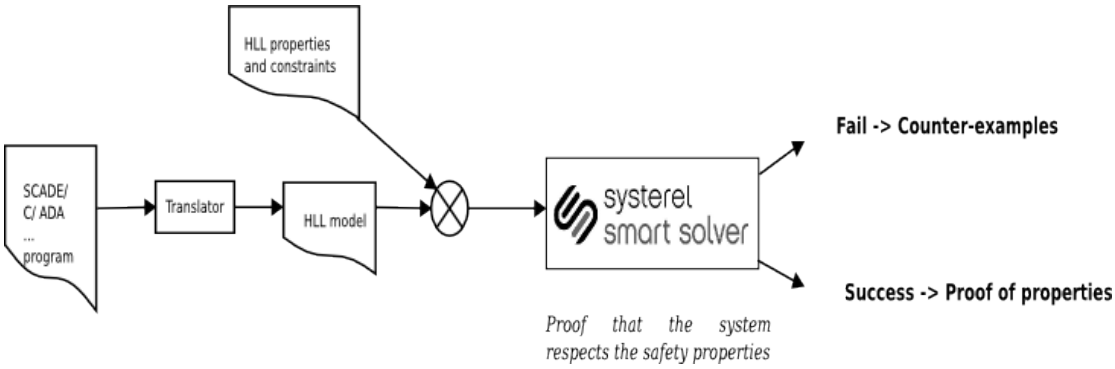


Figure 14. Safety Property Verification

First the source program in either C, SCADE, Ada,... is translated into HLL format via a translator; this HLL system model is then combined with safety properties expressed in HLL to form a verification model. Environment hypotheses and constraints can be added if a property does not hold based only on the formal model of the software and requires additional hypotheses about the overall system and/or its environment.

Finally the properties are verified using S3.

A good approach is to first use the BMC strategy with a rather large depth (depending on the system). After, if there is no counter-example, the induction strategy can be launched.

6.4.4 Application of S3: Equivalence Verification of Different Models

There are different areas where the formal verification of equivalence is required. One such example is the verification of equivalence of two different tool chains for the same task in an approach based on diversification. Such an approach is often used in the development of safety critical systems, to decrease the probability of errors. In general, two different versions of a software are developed independently, using different programming languages, different approaches and also separate teams.

The equivalence of the two resulting system models is then verified using S3.

To prove equivalence between two HLL models, we make the hypothesis that the inputs of the two HLL models are equal and we want to prove that the outputs are equal (see Fig. 15).

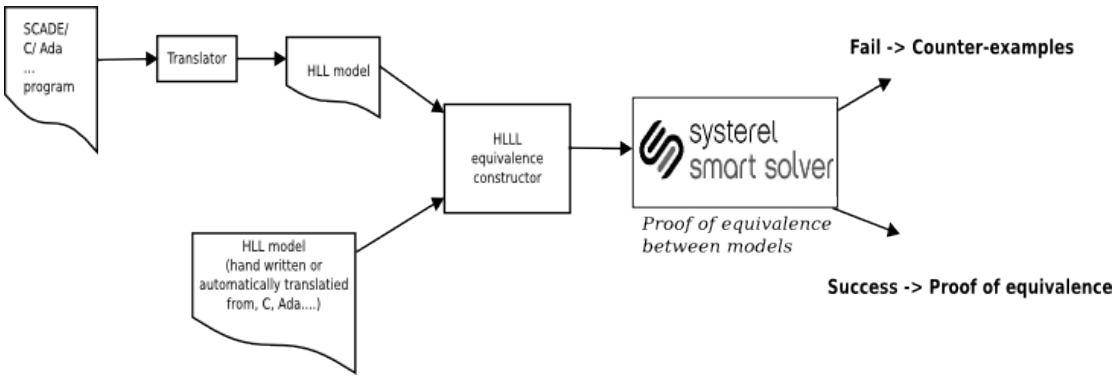


Figure 15. Equivalence Check

The equivalence could also be used to prove that a code is equivalent to a specification: the code is translated into HLL and the specification is written in HLL.

It is also used in the certification flow (see section 6.4.6).

6.4.5 Application of S3: Test Case Generation

S3 can also be used for test case generation:

- In the case of functional black-box testing, the main difficulty in writing of test scenario is to define the values of the outputs to observe as a function of the input values, as the analysis of the functionality can be complex.
- In the case of white-box verification, the difficulty is to define the right input values to cover a function, a branch or a condition.

As an alternative, for black box-testing, S3 shall allow to determine easily test oracle. For white-box testing, we can easily write a test objective in HLL that states exactly the objective of the test (e.g. the desired output values). Applying a BMC strategy, we obtain all the desired scenario with the expected values of the inputs.

6.4.6 Certifiable Systerel Smart Solver

In order to conform to industrial standards requirements for critical systems [**? ?**], we propose a certifiable formal verification solution with S3 (see Fig.16).

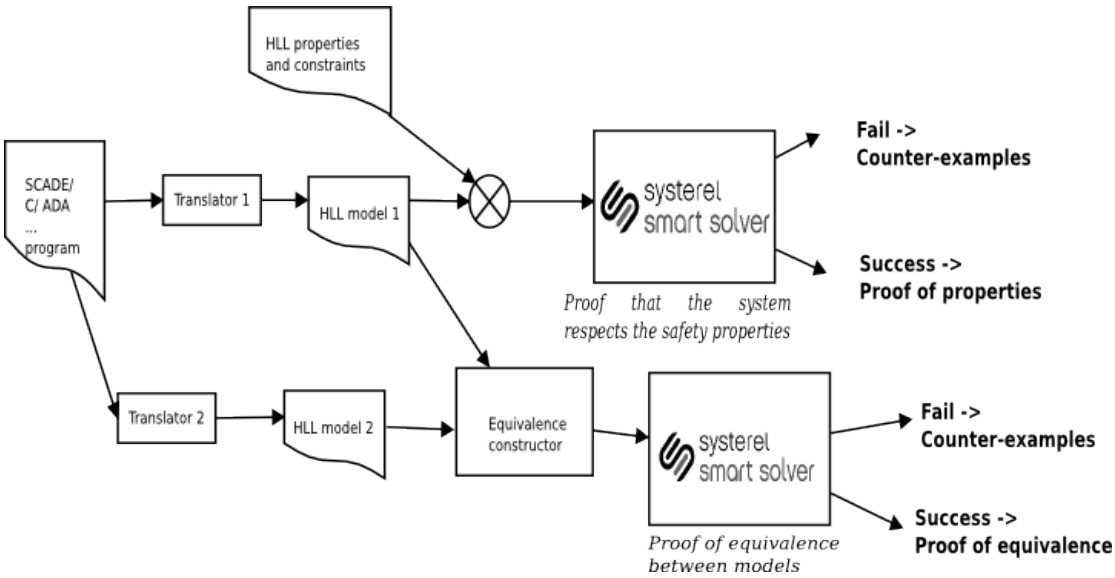


Figure 16. Certifiable solution

When building certifiable formal verification solutions, the certifiable Systerel Smart Solver (cS3) approach relies on three different techniques:

Diversification of the translation chain: the translation of a model to HLL is done twice[8], with two translators being developed by two independent teams in two independent programming language (for instance one in C the other in Ocaml).

Equivalence-check of the translated models: the outputs of the two diversified translation chains are compared using equivalence checking. S3 is used together with an equivalence-constructor to check if the two translated models are *sequentially equivalent*, *i.e.*, if given the same scenario on their inputs, they would produce the same outputs.

Record of results in a verifiable proof-log: when a proof is validated or an equivalence between model established, the result of the S3 is not a simple "OK" answer, but a *proof-log* file that contains an encoding of the proof of this claim in a sound and complete proof-system. An independent checker is run *a-posteriori* to check the correctness of this proof.

6.4.7 Application on Modes and Levels function

The approach has been applied on Modes and Levels function modelled in Scade ([9])

Name	Type	Model	SRS coverage	section
isolate	simple proof	Modes	4.6.2 C1	
no_power	simple proof	Modes	4.6.2 C4	
level case	use case	Level	5.10	
shunting initiated by driver	validation	Modes	5.6	
start of mission	validation	Modes	5.4	

[8] When applicable, these diversified translations are performed from differentiated sources.

[9] `https://github.com/openETCS/modeling/tree/master/model/Scade/System/ObuFunctions/ManageLevelsAndModes`

6.4.8 Application of S3: Static analysis

At first, the formal verification of the model was used to find bugs in the developed SCADE model. S3 adds some proof obligations to assess that the HLL model is correctly defined:

- Indexes of arrays belong to their ranges
- Latch definition range check
- No division by 0
- No overflow and no underflow on arithmetic expressions
- Output and constraint initialization check

Besides, the translators from a language to HLL can also generate proof obligations to be analyzed by S3, to check that the code does not have undefined behavior with respect to the source language.

For example the C-translator adds some proof obligations to ensure conformance with the C99 standard.

Results The three models have been verified on their topnode:

SCADE model	Top Node	Results
Modes	ManageModes	PO 1-5: valid
Levels	ManageLevels	PO 1-5: valid
ModesAndLevels	ManageLevelAndMode	PO 1-5: valid

6.4.9 Conditions to Isolate mode

Files used for the proof The proof is defined in the file `https://github.com/openETCS/validation/blob/master/VnVUserStories/VnVUserStorySysterel/05-Work/S3/Small_Proof/isolate.hll` and it is verified on the top node *ManageModes* of the SCADE model *Modes*.

What is proved ? The Condition 1 of SRS § 4.6 "The driver isolates the ERTMS/ETCS on-board equipment" is proved, ie; as soon as the input of SCADE model `Data_From_DMI.'ETCS_Isolated'` becomes true, the output `currentMode` becomes 'isolated' (`Level_And_Mode_Types_Pkg::IS`) and internal condition is activated.

Constraints used None.

Results The property is proved.

6.4.10 Procedure Shunting initiated by Driver

Files used for the proof The proof is defined in the file `https://github.com/openETCS/validation/blob/master/VnVUserStories/VnVUserStorySysterel/05-Work/S3/Proof_SoM/shunting_initiated_by_driver.hll` and it is verified on the node *Procedure_SH_Initiated_By_Driver* of the SCADE model *Modes Management*. The same proof is also defined in the file `https://github.com/openETCS/validation/blob/master/VnVUserStories/VnVUserStorySysterel/05-Work/S3/Proof_SoM/shunting_initiated_by_driver_topnode.hll` to be verified on the top node *ManageModes* of the SCADE model *Modes Management*.

What is proved ? We want to prove that the procedure SH_Initiated_By_Driver is a correct implementation of the section 5.6 Shunting Initiated By Driver of SRS-26.

To prove this, a specification of the flowchart is proposed in the property file. However, the flowchart is not entirely specified: elements `D030`, `A030`, `A095`, `S100` and `A115` of the flow chart in SRS-26 are out of the scope of the mode management function.

Constraints used One hypothesis is used in this model to avoid a counter-example: when we are in `A100` the request of "End of Mission" procedure correspond to the value of the input `On-going Mission` as it is specified in the SCADE model.

This hypothesis is justified by the fact that, according to `A050`, `D040` and `A100`, if the input `On-going Mission` is `True` then the "End of Mission" request is `True`, so equal to `On-going Mission`. Also, as transition to SH mode is enable (`A050`) when "End of Mission" request is made, the system should go to SH mode (or another mode except SB mode).

Results Considering the constraint and our model, the SCADE model of SH_Initiated_By_Driver corresponds to the specification. Proof of this property allow to detect an error in SCADE model: operator *AND* was used instead operator *OR*. Besides the specification in SCADE of the computation of the output *End_Of_Mission* was corrected.

6.4.11 Procedure Start of Mission

Files used for the proof The proof is defined in the file `https://github.com/openETCS/validation/blob/master/VnVUserStories/VnVUserStorySysterel/05-Work/S3/Proof_SoM/startofmission_topnode_proof.hll` and it is verified on the node *Procedure_StartOfMission* of the SCADE model *Modes*.

What is proved ? We want to prove that the procedure Procedure_StartOfMission is a correct implementation of the section 5.6 Procedure Start of mission of SRS-26.

Only the down part of the flowchart, from S10 and from S20, relative to the modes management, is specified in the SCADE model.

Constraints used

1. Level should not change : Level can be change at the beginning of Start of Mission procedure, but when we go in the step of mode selection, the level should not change.
2. Train Data should not change : Train Data are validated at the beginning of Start of Mission procedure, they shall be valid to start mode selection and we consider then that their validity should not change.
3. The train shall stay at standstill during all the procedure. Indeed in Stand-By mode, standstill shall be ensure by supervision function (see SRS-26 §4.4.7.1.5).
4. The "On Going Mission" variable, input of SH Initiated by Driver, is forced to False. This is justified by SRS 5, section "5.4.6 Entry to Mode Considered as a Mission".

Results Considering constraints and our HLL model, the SCADE model of Procedure Start of Mission corresponds to the specification. Some errors have been detected in the SCADE model: links between nodes were wrong.

6.4.12 Conclusion

Benefits of formal methods in an a posteriori verification process of critical systems has been recognized by our industrial customers:

- Contrarily to a human generated test-based verification solution, a formal safety verification is intrinsically complete. It is equivalent to search for every possible falsification.
- It clearly identifies the complete list of assumptions upon which the safety relies.
- A certified solution allows for a reduction of the testing and review efforts (only the generic safety specification has to be reviewed).
- The use of formal verification in the qualification of critical software sends a strong and positive message to the market, and is sometime even a requirement for some customers.

7 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.

7.1 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.

7.2 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 [**?**] 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.

7.2.1 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 17.

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 4. 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 4. Transitions between Running States

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

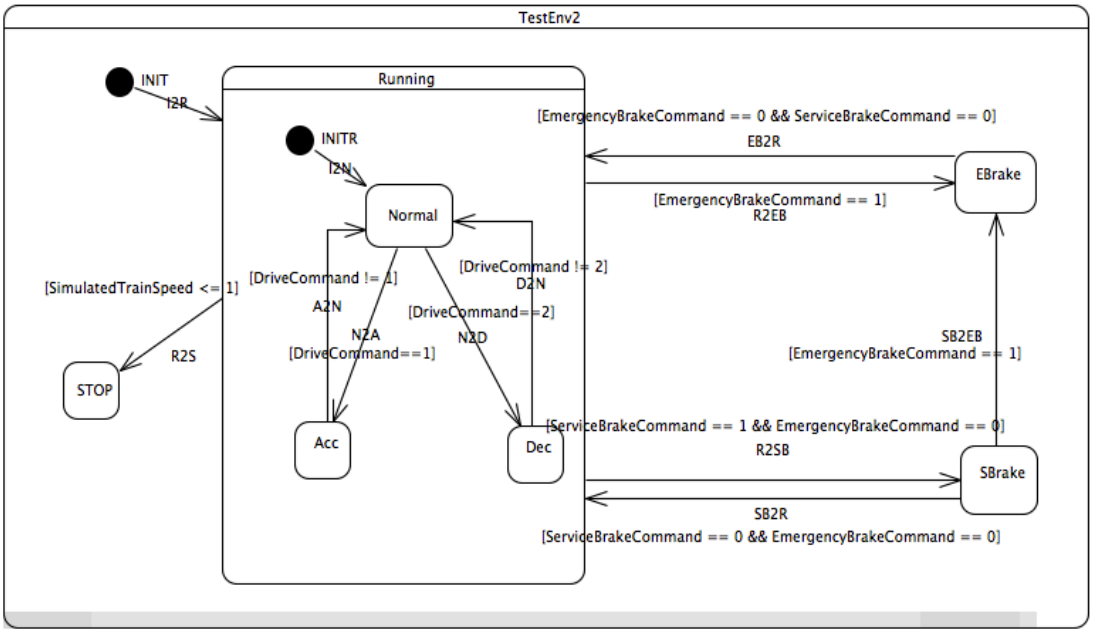


Figure 17. CSM Test Environment Model

The actions on each transition (the "do behaviors") are described in Table 5. 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 5. 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 6. The transitions are controlled by the commands EmergencyBrakeCommand and ServiceBrakeCommand generated from the train control system.

Table 6. 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 7. 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 7. Do Behaviors in Environment States

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

7.2.2 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. 18. This test describes a situation where the driver accelerates for a time T1 (DriveCommand = 1) before decelerating for a time T2.

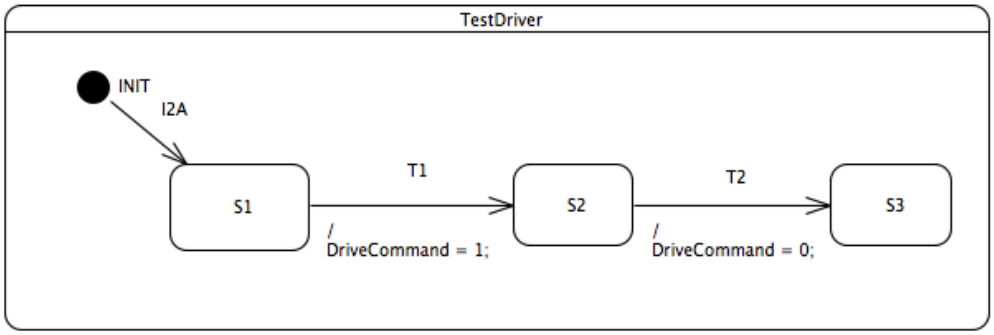


Figure 18. Test Driver Model

7.3 Detailed verification plan

7.3.1 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.

7.3.2 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 [**?**]. 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.

7.3.3 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 [**?**] 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.

7.4 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 8. 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 9. 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 [**?**]. 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 10. 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.

7.5 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.

7.6 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 10. 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)))

7.7 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.

8 Conclusion