Verification and Validation of Event-B models

Matthias Güdemann
Systerel

November 25, 2013

Abstract

This document describes the V&V processes applicable to Event-B models. It supports Event-B based on the Rodin platform as a very promising method in the context of the openETCS project. It is a rigorous formal method, based on the well-established B method, aimed at formal system analysis. Implemented in the Rodin tool, based on the Eclipse platform, it allows the usage of many existing plug-ins and allows for easy integration with new Eclipse plug-ins.

The Event-B modeling approach allows for a provably correct implementation of the requirements of SS 026, showing the degree of coverage of the requirements, formalizing and proving identified properties of the safety analysis and proving non-testable requirements of the test-case specification.

Contents

1 Introduction

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.

2 Verification Processes Applicable to Event-B

2.1 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.

2.2 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.

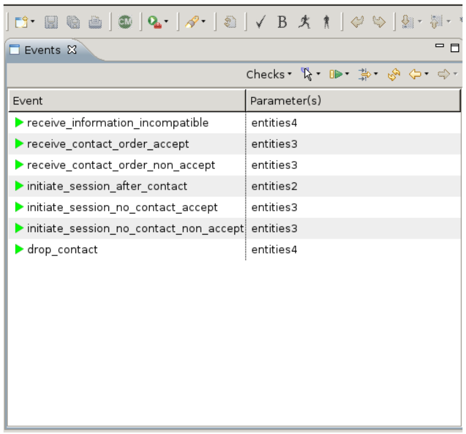
2.3 Model Simulation

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 1 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.

2.4 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:



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

Figure 1: ProB Model Animation

- **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.

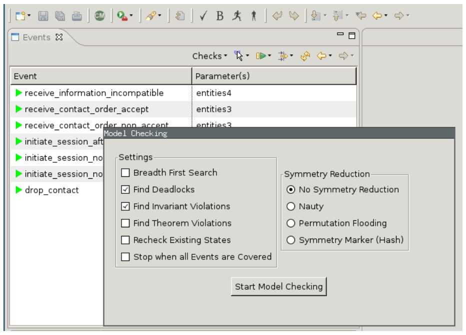
- **Deadlock Freeness** A deadlock represents a state where the system that the system cannot leave as no event is enabled. For a reactive system this is always an unwanted state that must be avoided.

- **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 correct 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 2 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



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

Model Checking

Settings

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

Symmetry Reduction

● No Symmetry Reduction
○ Nauty
○ Permutation Flooding
○ Symmetry Marker (Hash)

Figure 2: Model-Checking for Deadlocks

that leads to a situation where no event can be selected anymore.

2.5 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.

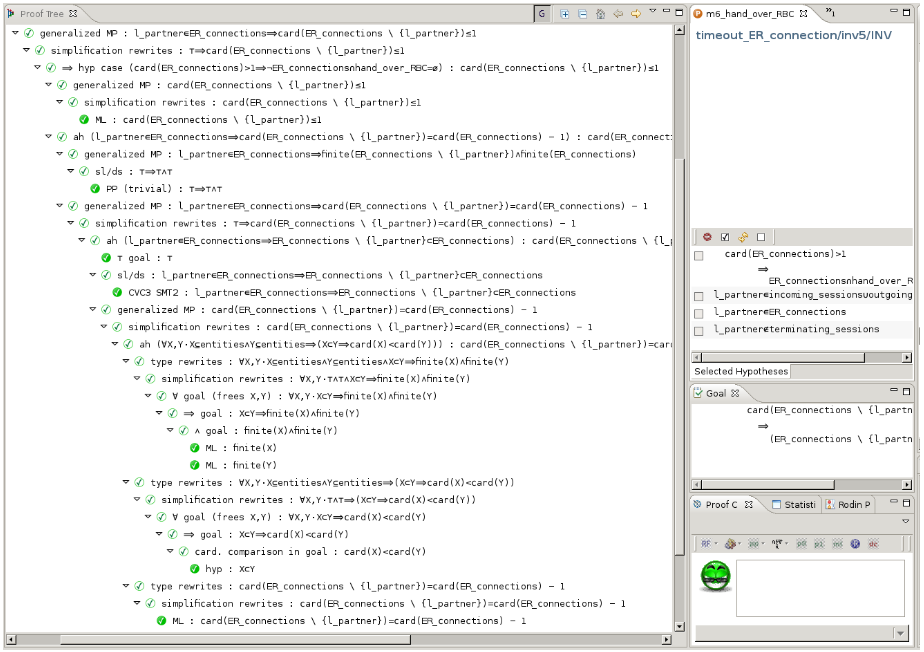
The Rodin platform natively supports manual construction of formal proofs by allowing easy access and manipulation of the proof tree and predicate hypotheses. It also provides access to different automated provers, i.e., the free of charge AtelierB provers, an open source SMT plug-in that supports several solvers[1] as well as an open source plug-in that connects Rodin to the Isabelle/HOL proof assistant.

Figure 3 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.

2.6 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.

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



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

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

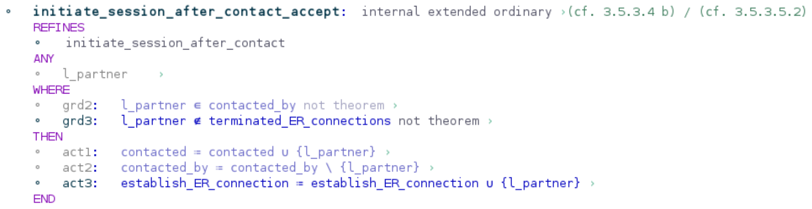
```
card(ER_connections \ {l_partn
  ⇒
    (ER_connections \ {l_partn
```

Figure 3: Rodin Proof Tree

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.

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

Figure 4 shows a refining event with guard strengthening and an additional



```
∘   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 4: Event Refinement

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.

2.7 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 [1]. 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 of the Event-B model with the ProR based on an EMF model of the Event-B model.

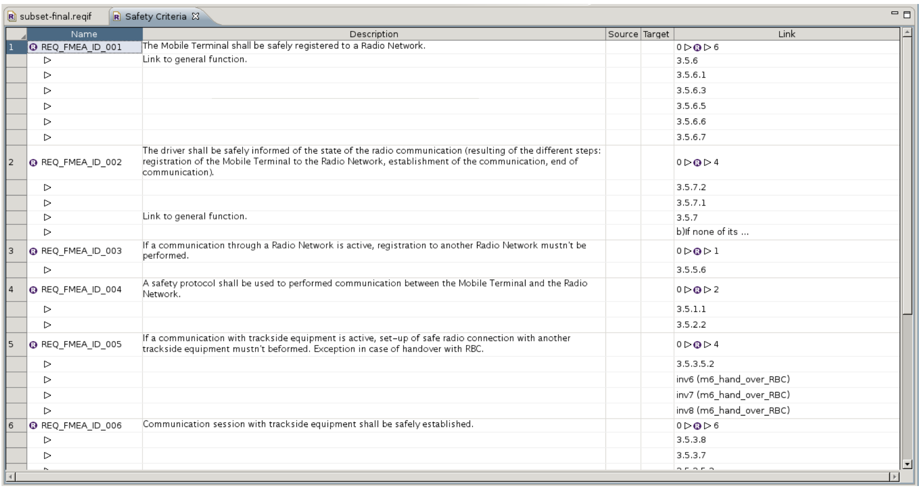
2.8 Verification of Safety Requirements

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 [2], 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 SS 026 section 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 5 shows a ReqIf document in Rodin (via the ProR plug-in) which holds the safety requirements defined by the safety analysis. For each requirement, there are references to the concerned elements of SS 026 and to Event-B



	Name	Description	Source	Target	Link
1	Ⓡ REQ_FMEA_ID_001	The Mobile Terminal shall be safely registered to a Radio Network.			0 ▷ Ⓡ ▷ 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 ▷ Ⓡ ▷ 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 ▷ Ⓡ ▷ 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 ▷ Ⓡ ▷ 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 ▷ Ⓡ ▷ 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 ▷ Ⓡ ▷ 6
	▷				3.5.3.8
	▷				3.5.3.7

Figure 5: Safety Requirements

elements where applicable, e.g., REQ_FMEA_ID_005 which is linked to the invariants, inv6, inv7 and inv8.

2.9 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 [1]. 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.

3 Object of Verification

The object of verification is the Event-B model for the communication establishing[2]. 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 SS 026, section 3.5[3].

[2] `https://github.com/openETCS/model-evaluation/tree/master/model/Event_B_Systerel/Subset_026_comm_session`

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

3.1 Detailed verification plan

3.1.1 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 SS 026 as well as proving correctness of the model wrt. the requirements ans 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.

3.1.2 Method/Approach

At first, the basic functionality described in the section 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 SS 026 are traced using links from Event-B to the ProR file in ReqIf format. Requirements are formalized as invariants and proven where applicable.

3.1.3 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

4 Results

- The result is a fully formal model of the communication session management as described in section 3.5 of SS 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 requirements are formalized as invariants in predicate logic, their proofs are for the most part fully automatic.
- It was found that while the SS 026 communication management explicitly allows multiple communication partners (see RBC handover), there is no explicit limit of established communication connections given in section 3.5.
- A complete covering of the elements of SS 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.

4.1 Summary

The created fully formal functional model allows for formalization and proof of SS 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[3] proof obligations, only 12, i.e., 3.2% require any manual intervention.

4.2 Evidence produced

The formal Event-B model, including a ReqIf document for section 3.5 of SS 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`

4.3 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.

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

The elements of SS 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.

4.4 Future Activities

For other sections of SS 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.

References

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

[2] Brice Gombault, *Safety analysis of Subset 026, Section 3.5, Management of Radio Communication (MoRC)*, version 4A, 14. 11. 2013, `https://github.com/openETCS/validation/blob/master/VnVUserStories/VnVUserStorySysterel/04-Results/a-SafetyAnalysis/safety_analyse_MoRC.pdf`

[3] UNISIG, *SUBSET-026 – System Requirements Specification*, version 3.3.0, 2012

End of Document