International Journal on Software Tools for Technology Transfer (2020) 22:315–332
https://doi.org/10.1007/s10009-020-00551-6

ABZ 2018

Validation and real-life demonstration of ETCS hybrid level 3 principles using a formal B model

Dominik Hansen[1,2] · Michael Leuschel[1] · Philipp Körner[1] · Sebastian Krings[1] · Thomas Naulin[2] · Nader Nayeri[2] · David Schneider[1] · Frank Skowron[2]

Published online: 15 February 2020
© The Author(s) 2020

Abstract

In this article, we present a concrete realisation of the ETCS hybrid level 3 concept, whose practical viability was evaluated in a field demonstration in 2017. Hybrid level 3 introduces virtual subsections as sub-divisions of classical track sections with trackside train detection. Our approach introduces an add-on for the radio block centre (RBC) of Thales, called virtual block function (VBF), which computes the occupation states of the virtual subsections using the train position reports, train integrity information, and the track occupation states. From the perspective of the RBC, the VBF behaves as an interlocking that transmits all signal aspects for virtual signals introduced for each virtual subsection to the RBC. We report on the development of the VBF, implemented as a formal B model executed at runtime using ProB and successfully used in a field demonstration to control real trains.

Keywords B-method · Animation · Model-based testing · Model checking · ETCS

1 Introduction

1.1 ETCS levels 1–3

The European Train Control System (ETCS) provides three progressively advanced levels for controlling trains. In ETCS level 1 the track is fitted with Eurobalises. When trains pass over those balises, they obtain precise position information and (statically precomputed) movement authorities. Optical signals are still necessary in level 1 and need to be obeyed by the train driver. Level 1 also requires trackside detection devices which detect whether a portion of track is free of any train. In practice this is done using either track circuits or axle counters. The latter count the number of axles entering and leaving a track section; if no axles remain, a portion of track is considered to be free.

In ETCS level 2 movement authorities are provided by a radio block centre (RBC) which communicates by radio with the trains. The trains send regular position reports to the RBC. These reports provide an alternate means to locate trains (with some delay and imprecision and obviously only for those trains that do send messages). While optical signals are no longer required, Eurobalises are still used as reference points for train positioning, and trackside detection is also necessary, e.g., to deal with non-talkative trains. ETCS

✉ Michael Leuschel
michael.leuschel@hhu.de

Dominik Hansen
dominik.hansen@hhu.de

Philipp Körner
p.koerner@hhu.de

Sebastian Krings
sebastian.krings@hhu.de

Thomas Naulin
thomas.naulin@thalesgroup.com

Nader Nayeri
nader.nayeri@thalesgroup.com

David Schneider
david.schneider@hhu.de

Frank Skowron
frank.skowron@thalesgroup.com

1 Institut für Informatik, Universität Düsseldorf, Düsseldorf, Germany

2 Thales Deutschland GmbH, Berlin, Germany

level 2 still operates with so-called fixed blocks, which are considered either completely free or fully occupied.

Trackside detection can be quite expensive and does not provide details about which trains are occupying a particular track section. In ETCS level 3 one can dispense with trackside detection: there no longer are fixed track sections that are marked as free or occupied. Instead, every train is surrounded by an envelope (moving block) which is reserved for just this train. An important concept in ETCS level 3 is train integrity: in principle a train can lose its integrity, e.g., lose some of its wagons. This can pose a risk to trains which follow such a train, in particular if there is no trackside detection. If a train is not guaranteed to have full integrity, it is thus not safe to release track sections behind the train. ETCS level 3 thus requires trains to have built-in train integrity detection, and may require "sweeping" the track if integrity is lost.

1.2 Hybrid level 3

The new specification "Hybrid ERTMS/ETCS Level 3" (HL3) [1] describes a novel train control concept, incorporating classical trackside train detection, radio-based position reports, and train integrity information.

Indeed, the absence of trackside detection in ETCS level 3 can lead to degraded performance when a train loses integrity or can no longer communicate with the RBC. The motivation of hybrid level 3 (HL3) is to combine the advantages of ETCS level 2 and 3. When present, existing trackside detection can be used by HL3 to deal with non-talkative trains or with situations where trains lose integrity. Also, HL3 does not use full moving blocks, but divides sections into virtual subsections. These virtual subsections act like fixed blocks, but with a much finer granularity than in ETCS level 2. This can be seen in Fig. 1: at the bottom there are two sections with trackside detection, which are divided each into three virtual subsections. The occupancy status of a virtual subsection is derived from train position reports in combination with trackside detection information. Hence, one can obtain more throughput (fit more trains on the track), while reusing ETCS level 2 equipment to a large extent. Also, in contrast to a solution without any trackside train detection (pure level 3), not all trains need to be equipped with an ETCS on-board unit and a TIMS (Train integrity monitoring system).

1.3 Virtual block function for HL3

In June 2017 the Heinrich Heine University Düsseldorf (HHU) was asked by Thales Deutschland GmbH to contribute to a field demonstration of feasibility of the ETCS hybrid level 3 principles. The call for tender was initiated by ProRail Netherland, with a demonstration planned on a test track at the ETCS National Integration Facility (ENIF), provided by Network Rail (UK) for December 2017.

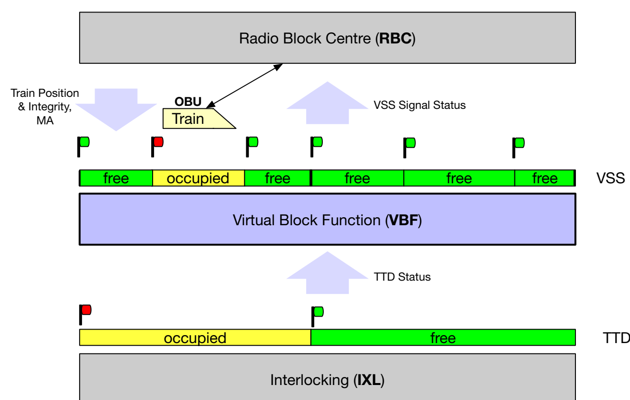

Fig. 1 The role of the virtual block function (VBF)

This resulted in the present cooperation between Thales and HHU, with additional support provided by ClearSy. The goal was to develop an executable version of the HL3 specification, called virtual block function (VBF), which is an add-on for the existing Thales radio block centre (RBC) without adapting the RBC core functionalities. The main idea is that the VBF partitions each trackside train detection section (TTD) into virtual subsections (VSS). For the RBC, the track is thus decomposed into finer-grained sections compared to the TTDs. The VBF computes the occupation status of each VSS by using the TTD occupation status and train position reports including train integrity information.

As mentioned, in Fig. 1 you can see that we have two areas at the bottom each with a trackside detection device. The VBF knows that the left one is occupied and the right one is free. However, for the RBC it simulates the existence of six areas and six trackside detection devices. Based on the train position information, the VBF can already free part of the occupied left track for following trains, enabling higher throughput without having to install additional trackside equipment. As far as the RBC is concerned, the VBF thus mimics an interlocking (IXL), providing an impression of a finer-grained track layout with additional trackside detection devices.[1]

Tools used System modelling was achieved using decomposition into components using the classical B syntax and structuring mechanisms (e.g., including or extending machines). The validation was done using the ProB animator and model checker [2], along with custom visualisations developed in JavaFX and Tcl/Tk. The final prototype was executed using the ProB animation engine, controlled by ProB Java API along with Java and C code to interface with the various trackside components at runtime.

[1] Note that an interlocking manages not just trackside detection devices and signals but also routes and points. In our initial experiments, the topology was linear, so there were no points and just one route to manage by the IXL.

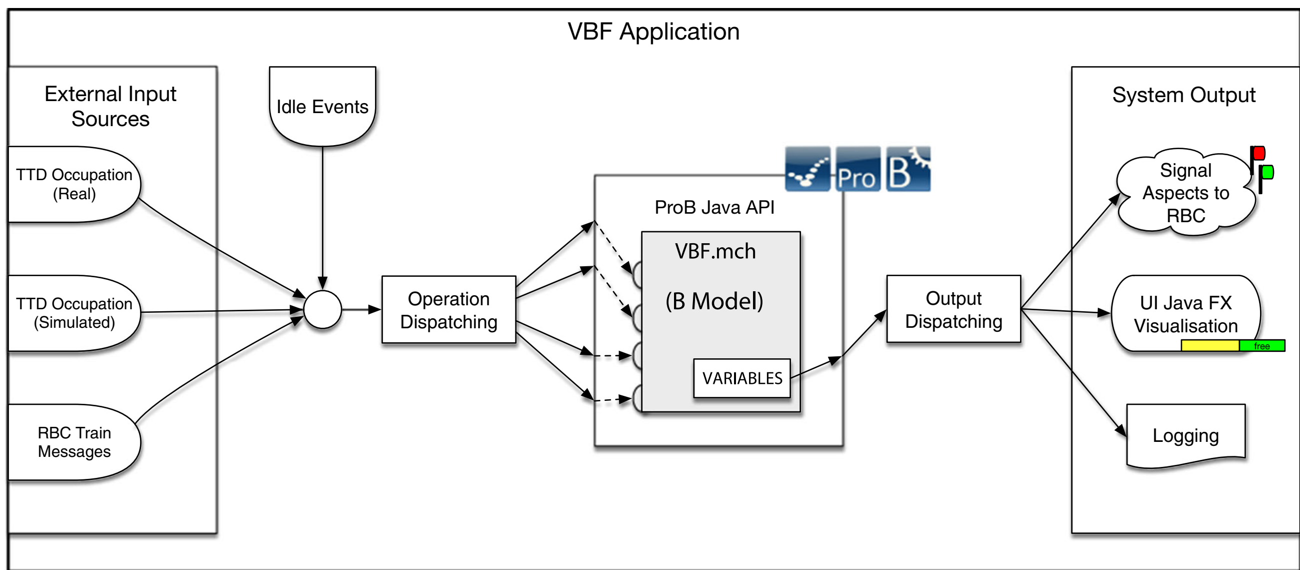

Fig. 2 Using the formal model for real-life demonstrations

Distinctive features The complete principles HL3 specification was modelled in a formal language. We uncovered several dozen important issues with the existing principles HL3 specification, leading to multiple changes which found their way into two new, updated versions of the reference document [1].

Note that the parts of the model shown in this paper and the issues that are discussed are based on version 1A of the HL3 specification.

The formal B model was executed without code generation, as illustrated in Fig. 2. It was run in real time using the PROB interpreter alongside either Thales simulators or real trains and interlockings. To our knowledge, this is the first time that a formal model was used *at runtime* as a prototype for a safety critical system. There are other uses of formal models as prototypes (see, e.g., [3]); here we use the formal model embedded in a real-world setting to conduct tests in real-time with actual hardware (real trains being controlled by the VBF in conjunction with real subcomponents such as the RBC and IXL and on-board units).

This is an extended version of the conference article [4], providing much more details about the modelling, the choices made and the validation conducted along with the issues found.

2 Formal language and tools used

The B-method [5] is a formal method that arose out of *Z* [6], with a focus on tool support and successive refinement to derive provably correct implementations out of high-level specifications. The B-method is arguably one of the industrially more successful formal methods. The initial industrial use of B was for line 14 in Paris [7], whose product has been adapted for many other metro lines worldwide (e.g., [8]). This initial version of B, as laid out in [5] and supported by ATELIER- B, is now called classical B or also “B for software”.

Out of the experience with classical B, Abrial developed a successor eventually called Event-B. The main addition though is a more flexible refinement concept targeted at systems modelling. Event-B is supported by the RODIN platform [9]. It is maybe less known that ATELIER- B also supports an Event-B dialect. Another less known fact is that one can also make use of classical B for systems modelling, using B’s inclusion mechanism to decompose a system into components. This is what we have done in this paper, and justify our choice later in Sect. 7.2.

In this paper we mainly use the animator and model checker PROB [2], which supports both classical B and Event-B, and also has support for the ATELIER- B dialect of Event-B. PROB also provides a few extensions to the core B language, which have not yet made their way into RODIN or ATELIER-B. In this paper we have used some of these extensions and also explain why we have done so.

2.1 Some background about B

A formal B model is composed of a variety of B machines. Each machine may contain any of the following items:

- new base sets.
- constants along with axioms (aka PROPERTIES) which describe the types and allowed values for the constants.
- variables along with invariants which describe the type and allowed values for the variables.

Table 1 Important B operators used in the model fragments

Symbol	Math	Meaning
!	$\forall$	Universal quantification
#	$\exists$	Existential quantification
&	$\wedge$	Conjunction
or	$\vee$	Disjunction
=>	$\Rightarrow$	Implication
=	$=$	Equal
/=	$\neq$	Not equal
:	$\in$	Set membership
/:	$\notin$	Not in set
<:	$\subseteq$	Subset
{}	$\emptyset$	Empty set
{x|P}	$\{x \mid P\}$	Set defined by predicate P
\/	$\cup$	Set union
/\	$\cap$	Set intersection
\	$\setminus$	Set difference
UNION	$\bigcup$	Quantified union
|->	$\mapsto$	Pair constructor
dom(r)		Domain of a function or relation r
ran(r)		Range of a function or relation r
f(x)		Application of a function f to x
size(s)		Length of a sequence
INTEGER	$\mathbb{Z}$	Set of mathematical integers
NATURAL	$\mathbb{N}$	Set of natural numbers (≥ 0)
POW(S)	$\mathbb{P}(S)$	All subsets of S
<->	$\leftrightarrow$	Relation
-->	$\rightarrow$	Total function
+->	$\nrightarrow$	Partial function
seq(S)		Set of sequences over S
iseq(S)		Set of injective sequences over S (i.e., no repetitions)

– operations which can change the values of the variables of the machine. Operations can take parameters and may have guards, i.e., predicates which state when they are enabled and can be executed. In the context of systems modelling operations are typically called events.

A machine may refine another machine, and may include any number of other subsidiary machines.

Table 1 contains the important B operators, that are required to understand the fragments presented in this article.

Here is a small B machine that manages trains and integrity information. The machine introduces two new base sets: `TRAIN` and `REPORTED_TRAIN_INTEGRITY`. The first one is called a deferred set: its members are left open and can be specified at a later point in time. The second one is fixed and called an enumerated set: all its members are provided. The machine has no constants but two variables along with invariants. The variable `registered` is a subset of `TRAIN` and the variable status is a total function from the registered trains to their integrity status. The operation `register_train` adds a new train to `registered` and sets its status to `no_integrity_information`. The operation is enabled when there exists a train `tr` which is not yet registered. The body of the operation is executed atomically, resulting in a final state where both `registered` and `status` are updated and the invariant holds.

```
MACHINE Trains
SETS TRAIN ;
     REPORTED_TRAIN_INTEGRITY =
     {confirmed_integrity,
      no_integrity_information,
      lost_integrity}
VARIABLES registered, status
INVARIANTS
  registered <: TRAIN &
  status : registered --> REPORTED_TRAIN_
INTEGRITY
OPERATIONS
  register_train(tr) = SELECT not(tr:registered)
 THEN
```

```
      registered := registered \/ {tr} ||
      status(tr) := no_integrity_information
  END
...
END
```

3 Core concepts of HL3

In HL3 a track consists of various sections, each with their own train detection device (TTD). Each TTD can either be *free* or *occupied*. Note that a faulty TTD can mark a track section to be occupied, even though it is in fact free. The converse is assumed to be impossible.[2]

Each TTD is divided into one or more virtual subsections (VSS). A VSS can have four different states, which describe its occupancy status:

- *free*: the VSS is guaranteed to be free;
- *occupied*: the VSS is occupied by a communicating train and nothing else;
- *unknown*: the VSS is not occupied by a communicating train, but may be occupied by other non-communicating trains or obstacles;
- *ambiguous*: : the VSS is occupied by a communicating train, and may also be occupied by other non-communicating trains or obstacles.

The HL3 specification describes 12 (4×3) transition of a VSS state to another state. Each transition is divided into various cases (#1 A, #1 B, …), and each case has various conditions associated with it. Table 2 shows the two variations of transition 9 from ambiguous to free and the associated conditions. Observe that some transitions have priorities over other transitions. For example, the transition #9B has priority over all transitions 10 from ambiguous to unknown: if the conditions for #9B and any variant of #10 are satisfied, only transition #9B will be performed.

The transitions take into account changes in the TTD status, new position reports from the trains, but also various timers which can expire. For example, there are timers which expire when a train has not sent a position report for too long. More details about HL3 can be found in the introduction to this special issue. In particular the precise notion of ghost trains and shadow trains, and the associated timers. Paraphrasing [1], a ghost train is either a real physical object or a train occupying a TTD, but which is unknown (i.e., it is not sending position reports). It may also be a virtual artefact, which seems to occupy a TTD, but is caused by a failure of the TTD. A shadow train is a ghost train that is following a known train.

[2] Each TTD is a SIL4 device, i.e., on average one dangerous failure per 10,000 operating years.

4 Requirements and modelling strategy

Due to the strict deadline for the HL3 field demonstration (in December 2017) and the very short time span for the project, it was decided to use off-the-shelf RBC and interlocking systems and use a **formal B model** [5] of the VBF as an **executable demonstrator**. More precisely:

- The Thales RBC core was to be used as is, without modifications for HL3. (Thales owns a product line for the RBC software to configure the generic software to the project specific requirements).
- The interlocking was used as is, without modifications for HL3.[3]
- The VBF had to be developed from scratch as an add-on for the RBC, which was to mimic an interlocking and transmit the signal aspects for the virtual signals to the RBC. The VBF contains a VSS state machine, with four possible states (free, occupied, unknown and ambiguous) for each VSS, exactly as required by the HL3 specification.

The following main tasks are the focus of this project:

T1: Providing evidence that the HL3 principles are consistent and complete to handle possible hazards and to allow the desired operational behaviour.
T2: Implementation of the VBF as an independent software unit by supporting the given interfaces to the other components. The implementation should conform to the HL3 principles.

To accomplish the first task, we decided to derive a formal B model from the HL3 specification. The decision was based on diverse work (e.g., [7,10–15]) which provided evidence that B is well suited for the railway domain. Moreover, first experiments were very promising: in a few days it was possible to model some simpler transitions of the HL3 specification.

For task T2, we intended to implement all interfaces (boundaries) to other components by hand and to use a classical testing approach to ensure their correct functioning. To reuse the formal model from task T1 for task T2, we had three options:

1. Using the model as a template to implement the VBF core by hand.
2. Generating code from the model and combine this code and the handwritten boundaries.
3. Executing the model at runtime by incorporating the execution engine and the handwritten boundaries.

[3] Except for the TTD occupation status which has to be sent from the IXL to the VBF/RBC.

Table 2 Definition of transition 9 from ambiguous to free in [1]

#	Condition	Priority over	Section ref.
#9A	(TTD is free)		3.1.1.5
#9B	(integer train has reported to have left the VSS) AND (the shadow train timer A of the TDD was not expired at the moment of the time stamp in the position report)	#10	4.5.1.7

The first option would require us to maintain both the model and the code. This could be time-consuming if there were changes to the specification (due to feedback from Pro-Rail, the specification was changed considerably). With the second approach, we would have to use an existing code generator (there was no time to develop our own) and thus have to refine our abstract B model down to implementation level B0—also time-consuming. Concerning the third option, we had already gained some experience of integrating ProB [16] as the execution engine in different software products [17,18]. Given our time constraints, the third option was the only feasible option, but it also posed the biggest research challenge: using a formal model at runtime interacting with various hardware and software components.

Model structure The VBF model (without environment) consists of 13 B Machines, 14 definition files and has 45 constants and 33 variables.

We used package pragmas to put topology independent parts separate from test data and from tests. We used ProB's package pragmas, which allow different machines to be put into different folders. The contents of a package `X` can be found in the folder of the same name `X`, while the contents of a package `X.Y` can be found in the folder `Y` within the folder `X`.

The main, topology independent machines were found in the main package and the associated `main` directory:

```
/*@package main */
MACHINE VSS_Constants
...
```

The environment model, still topology independent, was in another directory, `test/environment`:

```
/*@package test.environment */
/*@import-package main */
MACHINE ENV_Model
SEES
  VSS_TTD_Constants
 ...
```

Finally, the topology dependent machines were to be found various sub-folders of `test/environment`, such as `test/environment/ENIF` for the on-site field tests:

```
/*@package test.environment.ENIF */
/*@import-package test.environment */
/*@import-package main */
MACHINE ENV_Model_ENIF_LeftToRight
EXTENDS
  ENV_Model
 ...
```

We have various instances of our model in these sub-directories:

- a very simple topology,
- the topology from the HL3 specification (HL3),
- the actual topology to be used for the on-site field tests (ENIF).

The topology data was put into XML files, which were read using ProB's external library for reading XML. We reused parts of our Rubin engineering rules [17] to process the XML and turn it into B datavalues.

Traceability The names of the VSS state machine transitions have been used within the model:

- as the name of B operations. For example, the transition 9A from ambiguous to free is modelled by the B operation `VSS_Ambiguous_Free_9A`.
- some higher-level operations, which collect updates to the various VSS, return the track name and transition number (such as 9A) as return result, which can be inspected in the ProB animation interface.

We also used ProB's pragmas to include text from the HL3 specification, sometimes verbatim, along with elaborations/descriptions. This text is available when reading the model, but also in ProB's user interface. Indeed, ProB allows using two kinds of pragmas that help with traceability:

- label pragmas that precede predicates. This is useful for requirements identifiers or links to sections in the requirements document. The syntax of this kind of pragma is:

  `/*@label "LABEL" */` *Predicate*.

- description pragmas that follow predicates and identifiers. This is useful for longer justifications or explana-

tions. Such a pragma looks like the following:

Formula `/*@desc "DESC" */.`

The labels are shown by ProB e.g., in the state view. Also, when invariant violations occur, ProB can give the user the label of the false predicates. Descriptions can be shown by right-clicking on elements in the ProB animator.

An example use of labels is visible in the following excerpt from our model, to which we will return to later:

```
DEFINITIONS
 Guard9A(vss) == vss:VSS & vss_state(vss) =
 ambiguous
  & /*@label "(TTD is free)" */
    ttd_state(vss_ttd(vss)) = free
...
OPERATIONS
 VSS_Ambiguous_Free_9A(vss) =
      SELECT Guard9A(vss) THEN  ... END
 ...
```

5 Model details

Below, we present different aspects of our B model along with some source code snippets.

5.1 Basic datatypes

The modelling of the track was relatively straightforward, which is not surprising since B's relations can be used to represent graphs and B provides many convenient operators on relations and functions, which are just a special case of graphs (see, e.g., Chapter 14 of "Modeling in Event-B" [19]).

However, for pragmatic reasons, we did not use Event-B [19] but rather classical B [5] for modelling the VBF. For example, we identify the VSSs, TTDs and trains using classical B strings. For simulation and execution purposes, we had to read topology and configuration data from XML files and thus could not populate enumerated sets beforehand. The conversion of the XML file into B data structures for the VBF model is also done in classical B using records and strings.[4] Finally, we have used other features, such as machine composition and operation calls (see Sect. 5.2), not readily available in Event-B. More details can be found in Sect. 7.2.

Below, we try to give a flavour of our modelling by showing some derived data structures for the track topology. First, let us examine a part of a topology independent B machine in our development.

```
PROPERTIES
  VSS : POW(STRING)
& TTD : POW(STRING)
& VSS /\ TTD = {}
& next_vss : VSS +-> VSS
& vss_ttd: VSS --> TTD // maps VSS to their TTD
& TTD_STATE = {free,occupied}
           // TTDs only have these two states
& next_ttd : TTD +-> TTD
& last_vss: TTD --> VSS
& /*@label "the last vss is part of its TTD" */
  !t.(t:TTD => vss_ttd(last_vss(t)) = t)
& /*@label "a successor of a last vss is
            in another TTD" */
  !(t,n).(t:TTD & last_vss(t)|->n : next_vss
           => vss_ttd(n) /= t)
...
```

For example, the `next_vss` constant is a partial function which links VSS to their successor VSS. The direction of the track is thus constant for any given execution run.[5] However, the direction of the track can be toggled after executing a scenario, since the conversion of the XML data is parameterised. Below we show how the `next_vss` constant is derived in another B machine. Observe that we allow the IF-THEN-ELSE to be applied to expressions and use an external B function (see Section 6.3 in [17]) to read in the track data from an XML file.

```
PROPERTIES
  TRACK_DATA = READ_XML("./resources/prj_ENIF_
01@STR.xml")
...
& C_VSSSequence = DeriveVSSSequence(TRACK_DATA)
...
& next_vss = UNION(i, ii).(
   i : dom(C_VSSSequence) &
   ii : dom(C_VSSSequence) & ii = i + 1
   | {IF RUNNING_DIRECTION = "LEFT_TO_RIGHT"
      THEN C_VSSSequence(i)  |-> C_VSSSequence
(ii)
      ELSE C_VSSSequence(ii) |-> C_VSSSequence
(i) END
     } )
 ...
```

5.1.1 Train status

Modelling the integrity state of trains revealed some ambiguities and inaccuracies within the HL3 specification. The concept "integer" (for a train) is used in different contexts within the specification. We try to explain the differences with the aid of our model:

```
SETS
   REPORTED_TRAIN_INTEGRITY =
     {lost_integrity, confirmed_integrity,
```

[4] The conversion is not shown in this paper since the XML data format is proprietary.

[5] Every scenario in the HL3 specification only has a single linear track with trains running in one direction. Points are not considered by the current version of the HL3 specification and they were not required for the field tests at ENIF.

```
      no_integrity_information}
 ; INTERNAL_TRAIN_INTEGRITY = {integer, not_
integer}
PROPERTIES
 TRAIN_INTEGRITY_MAPPING = {
  "TRAIN_INTEGRITY_CONFIRMED_BY_INTEGRITY_
MONITORING_DEVICE"
        |-> confirmed_integrity,
  "TRAIN_INTEGRITY_CONFIRMED_BY_DRIVER"
        |-> confirmed_integrity,
  "NO_TRAIN_INTEGRITY_AVAILABLE"
        |-> no_integrity_information,
  "TRAIN_INTEGRITY_LOST"
        |-> lost_integrity}
...

INVARIANT
  registeredTrains : POW(STRING) &
& train_reportedTrainIntegrity
          : registeredTrains --> REPORTED_TRAIN
_INTEGRITY
& train_integrity
          : registeredTrains --> INTERNAL_TRAIN
_INTEGRITY
...
```

According to the ERTMS/ETCS specifications [20], a train can send four possible integrity status values within a train position report, which are represented by the domain of the constant TRAIN_INTEGRITY_MAPPING. However, the HL3 specification only distinguishes between three values, which are represented by the enumerated set REPORTED_TRAIN_INTEGRITY. The surjective function TRAIN_INTEGRITY_MAPPING defines the respective mapping bridging the gap between these two specification documents. Note, that this is not the only possible mapping as an infrastructure manager could require to map TRAIN_INTEGRITY_CONFIRMED_BY_DRIVER to no_integrity_information to exclude human failure.

Moreover, the HL3 specification defines, in Sect. 3.5, a further integrity state by using the terms "integer" and "not integer" which is represented by the enumerated set INTERNAL_TRAIN_INTEGRITY in our model.[6] The conditions of the state machine in the HL3 specification are referring to this integrity state. Yet, an unambiguous mapping from the reported train integrity to the internal train integrity is missing in Sect. 3.5. Thus, we were forced to find a sensible interpretation; we defined the following two conditions as triggers for the transition from "integer" to "non-integer":

– *"train reports 'lost integrity' "*
– *"PTD [Positive Train Detection] with no integrity information is received outside of the integrity waiting period"*

[6] The term "internal" train integrity refers to the internal state of the VBF.

Both conditions are part of the transitions #7B and #8A [1, Section 5.1.1.6]. The change of the train length (the remaining condition of #7B and #8A) does not affect the internal integrity status of a train but can have a consequence for VSS states as it triggers the "train integrity propagation timer" of the VSSs where the train is located.

The following operation manipulates the internal train integrity variable in our model:

```
 Train_SetIntegrityStatus(train, status) =
   PRE status : REPORTED_TRAIN_INTEGRITY
   THEN
     train_reportedTrainIntegrity(train) :=
status ||
     IF status = lost_integrity
     THEN train_integrity(train) :=
not_integer
     ELSIF status = confirmed_integrity
     THEN StartTimerDelta(train|->WAIT_
INTEGRITY_TIMER)
           || train_integrity(train) :=
integer
     ELSIF // no information available
       train |-> WAIT_INTEGRITY_TIMER :
expiredTimers
     THEN train_integrity(train) := not_
integer
     END
   END
```

However, the model checker ProB directly reported an invariant violation. This is because a train does not register itself by a train position report; thus, the variable train_reportedTrainIntegrity is not a total function with the registered trains as its domain. As a consequence, we had to make a further decision by treating a train as non_integer before the VBF receives the first position report (interpretation to the safe side). We always tried to avoid partial functions as it would mostly introduce handling of special cases. Moreover, the description in the HL3 specification is imprecise regarding when to start the first "wait integrity timer": *"A 'wait integrity timer' runs continuously for every train [...]"* [1, Section 3.4.1.3.1]. We decided to start the timer with first train position reported but not with the registration.

We found a further inaccuracy with regard to the integrity status in the specification: *"For an integer train the confirmed rear end location of the train is derived from [...]"* [1, Section 3.3.3.1]. Here, the term "integer train" is used which corresponds to the internal train integrity of our model. However, in Section 3.3.3.4 it is stated that *"the confirmed rear end of the train location is never updated by position reports with integrity status 'Lost' or 'No information available"'* [1, Section 3.3.3.4]. Thus, Section 3.3.3.1 of the specification should rather start with *"For a train which reports confirmed integrity"* since a train can be integer while reporting "No integrity information available".

5.1.2 Train location

Another essential concept in HL3 specification is the definition of the train location (in our case the image of the train location seen by the VBF) which is frequently referred within the state machine transitions of the HL3 specification. We mapped each registered train to a set of VSS within our model:

```
INVARIANTS
  ...
  & train_location : registeredTrains -->
POW(VSS)
  ...
```

In most cases, we just want to know if a certain train is located on a certain VSS. For these cases, the data structure for `train_location` as a total function from registered trains to sets of VSS is very convenient. Alternatively, we could have used a relation from trains to VSS, but we generally preferred functions over relations. One exception is the `next_vss` constant binary relation between VSS, which is frequently inverted in our model and can be obtained using B's relational inverse operator: `next_vss~`.[7] The order of the VSS associated to a train is not incorporated into the `train_location` definition, as this information is already contained in the `next_vss` constant relation. The condition that a train location must not have any gaps (which is not explicitly mentioned in the HL3 specification) can also easily be expressed with the aid of `next_vss` as follows:

```
   /*@label The train location must not have
any gaps */
    !loc.(loc: ran(train_location)
      => #s.(s : iseq(loc)
            & !i.(i : 1..size(s-1)
                     => s(i) |-> s(i + 1) :
next_vss)))
```

More precisely, we state that that for every train location `loc` corresponding to a set of VSSs, there must exist an ordering of the VSSs (i.e., an element of `iseq(loc)`) such that these are chained together by the next relation (i.e., `next_vss`). Remember that Table 1 contains a brief explanation of the B operators used in this article.

While the modelling of the train location data structures was relatively straightforward, the updates to this variable are, in our opinion, the most underspecified part of the HL3 specification. Some issues referring to the location are:

- Minor: *"As long as the TTD where the max safe front end is reported is free, the train location is not extended onto the VSS which are part of this free TTD"* [1, Section 3.3.2.1.2]. This is imprecise as the condition should be: only if the max safe front is reported to be on the next free TTD but not the estimated front of the train.
- Fundamental: *"[…] the train location is derived from the estimated front end […] of the last position report […] as well as from TTD information […]."* Is the train location only updated/changed by processing train position reports (in this case the TTD information will of course be considered)? Or does a single TTD change event without a train position report also update the train location? We had tried both alternatives and in the end we decided to use a train position report as the only trigger to update the train location (as used by the first demonstrator in December 2017). The other alternative would have forced us to adapt several transitions in order to be able to replay all scenarios of the HL3 specification (version 1A). Note, that the authors of the HL3 specification then decided on the other alternative in version 1B which was released in April 2018. As it was a fundamental change, it resulted in a lot of inconsistencies between the state machine and the scenarios which were not solved until version 1C (also thanks to our comments). But in the end, we think the authors' decision was the right one.

[7] Inverting a function in `VSS --> POW(VSS)` is more cumbersome in B.

5.2 State machine transitions and priorities

We decided to model each VSS state machine transition in the HL3 specification document as a B operation. For each variation of the 12 possible transitions we have one individual B operation, with at least the affected VSS as parameter. Some events also have the affected train as parameter.

The HL3 specification contains a table summarising all 12 transitions and their conditions. Table 2 shows the conditions for the variations of transition 9 from ambiguous to free. Below, we show the B translation of the state machine transition (#9A) of the HL3 specification.

```
MACHINE VSS_StateMachine
...
DEFINITIONS
 Guard9A(vss) == vss:VSS & vss_state(vss) =
ambiguous
  & /*@label "(TTD is free)" */
    ttd_state(vss_ttd(vss)) = free
...
OPERATIONS
VSS_Ambiguous_Free_9A(vss) =
  SELECT
    Guard9A(vss)
  THEN
    vss_state(vss) := free ||
    // state of the virtual signal which
protects the vss
    vss_signalState(vss) := PROCEED ||
    ...
  END
 ...
```

Observe that the operation has the VSS under consideration as parameter. Our model contains an outer loop which ensures that all enabled transition operations for all VSSs are processed; see Sect. 5.2.

For early validation (see, e.g., Sect. 6.3.1) it was useful to have an individual operation for each transition. In PROB's animation view we can see at a glance which transitions are enabled, and by inspecting the guard predicates we can analyse the reason a transition is enabled or not enabled (see traceability in Sect. 4).

However, in the model of the VBF, we need to execute in a given cycle **all** enabled VSS transitions, until no more are applicable. We assume that the order of processing the VSS updates is not significant (see [21] which studies just this question). We also need to encode the priorities, i.e., execute the transitions with a higher priority first. We have experimented with various ways of encoding the priorities, and have finally pursued a solution based on using a large IF-THEN-ELSE with the guards as conditions, calling respective operations of a subsidiary machine. More precisely, to solve this issue, we used the following approach:

1. extract the guard of each transition into a B definition, so that we can reuse it. Above you see the definition `Guard9A` which captures the conditions under which transition 9A is applicable.
2. write a B machine (`VSS_Statemachine_Step`) which includes `VSS_StateMachine` and which encodes the priorities using a nested IF-THEN-ELSE statement using the guard definitions.
3. write a B machine (`VSS_Statemachine_Run`) which includes `VSS_Statemachine_Step` and executes all update steps until completion using a WHILE loop.

The IF-THEN-ELSE in step 2 ensures that the priorities of the transitions are respected, e.g., that transition `2A` has priority over `3`, as requested by [1]. A return variable `out` stores the exact VSS transition taken for debugging and analysis. Here we show part of the corresponding operation:

```
MACHINE VSS_Statemachine_Step
INCLUDES VSS_Statemachine
...
out <-- VSSUpdateStep(vss) = PRE vss : VSS
  THEN
    IF Guard1A(vss) THEN
       VSS_Free_To_Unknown_1A(vss) || out :=
"1A"
    ELSIF Guard1B(vss) THEN
       VSS_Free_To_Unknown_1B(vss) || out :=
"1B"
    ...
    ELSIF #train.(train : registeredTrains &
                  Guard11B(vss, train)) THEN
        ANY train
        WHERE train : registeredTrains &
              Guard11B(vss, train)
        THEN
          VSS_Ambiguous_Occupied_11B(vss,
train)
          || out := "11B"
        END
    ELSE
      out := "NONE"
    END
  END
 ...
```

Execution of all VSS updates in a VBF cycle is done by a B WHILE loop calling `VSSUpdateStep`.[8] Again, the output `changes` is for debugging with PROB, to be able to see the VSSs that have changed their status and which HL3 transition was applied to perform the change.

```
MACHINE VSS_Statemachine_Run
INCLUDES VSS_Statemachine_Step
...
changes <-- VSSUpdateRun =
    VAR vssSet IN
      vssSet := VSS; changes := [];
      WHILE vssSet /= {} DO
        VAR currentVss, rule IN
          currentVss := CHOOSE(vssSet);
          vssSet := vssSet \ {currentVss};
          rule <-- VSSUpdateStep(currentVss);
          IF rule /= "NONE" THEN
            changes := changes <- (currentVss
|-> rule)
          END
        END
      INVARIANT vssSet <: VSS
      VARIANT card(vssSet)
      END
    END
 ...
```

5.3 Modelling of time

We had a dedicated B machine (GenericTimer) to keep track of the timers and the current time, modelled as an integer B variable `currentTime`. The progress of time is modelled by calling an operation `UpdateCurrentTime` which obtains the new time as an input and also updates the set of expired and running timers. It is the only way time can be modified in our model. During actual execution of our B model, the surrounding Java program was responsible for regularly calling the UpdateCurrentTime operation with the current system time in milliseconds as parameter.

The GenericTimer machine provides various facilities to add new timers and to inspect the status of the existing timers.

[8] CHOOSE is the Hilbert operator which returns a designated element for any given set. Note that CHOOSE is deterministic and is a mathematical function, i.e., it will always return the same element given the same set. It is an external function provided by PROB.

It is also possible to query the machine for the earliest deadline.

```
/*@package main */
MACHINE GenericTimer(TimerNameDomain)
VARIABLES
  timerDomain,
  timerDeadline,
  timerDefaultDelay,
  expiredTimers,
  runningTimers,
  currentTime
INVARIANT
  timerDomain <: Domain &
  timerDeadline : timerDomain --> INTEGER &
  timerDefaultDelay : timerDomain --> INTEGER &
  currentTime : NATURAL &
  expiredTimers <: timerDomain &
  runningTimers <: timerDomain &
  runningTimers /\ expiredTimers = {}
DEFINITIONS
  ACTIVE_DEADLINES == ran({at,dd | at:
timerDomain &
    dd=timerDeadline(at) & dd >= 0 &
currentTime < dd})
  ...
OPERATIONS
  AddTimerFor(d,default) =
  PRE d:Domain & d/: timerDomain & default:
INTEGER THEN
     timerDomain := timerDomain \/ {d} ||
     timerDeadline(d) := NOT_RUNNING ||
     timerDefaultDelay(d) := default
  END;
  ...
  UpdateCurrentTime(newCurTime) =
    PRE newCurTime>=currentTime
    THEN
          currentTime := newCurTime
      ||  expiredTimers :=
          {t|t:timerDomain & TIMER_EXPIRED(t,
newCurTime)}
      ||  runningTimers :=
          {t|t:timerDomain & TIMER_RUNNING(t,
newCurTime)}
    END;
  res <-- EarliestDeadline =
      IF ACTIVE_DEADLINES = {}
        THEN
          res := NOT_RUNNING
        ELSE
          res := min(ACTIVE_DEADLINES)
      END
...
```

6 Validation and verification

6.1 Animation of scenarios

The HL3 document describes a number of scenarios in addition to the VSS state machine. We used these scenarios as test specifications, i.e., to check that these scenarios are feasible in our model.

To animate the scenarios with ProB, we developed an environment model and composed it with the VBF core model (software model) to obtain a system model. The environment model has knowledge of the "real" (physical) position of a train, which allows it to move the train and to send train positions reports which are inputs of the VBF. Below we show some parts of this model. The invariant shows the relevant "real" state of the environment, as opposed to the view of the VBF (and the VSS statemachine).

```
/*@package test.environment */
/*@import-package main */
MACHINE ENV_Model
...
INVARIANT
   ENV_train_estimatedFEPosition : STRING +->
 INTEGER
 & ENV_train_length : STRING +-> NATURAL
 & ENV_ttd_state : TTD --> TTD_STATE
 & ENV_time : NATURAL
 ...
OPERATIONS
ENV_moveTrain(train, meter) =
  SELECT
    train : dom(ENV_train_estimatedFEPosition)
    & meter : {10, 100, 500}
  THEN
    ENV_train_estimatedFEPosition(train) :=
     IF RUNNING_DIRECTION = "LEFT_TO_RIGHT"
THEN
      ENV_train_estimatedFEPosition(train) +
meter
     ELSE
      ENV_train_estimatedFEPosition(train) -
meter
     END
    ; ENV_SET_TTD_STATE
  END;
out <-- ENV_Train_register(train) =
  SELECT
    train : dom(ENV_train_estimatedFEPosition)
    & train /: registeredTrains
  THEN
...
```

Figure 3 shows a system state where the "real" position differs from the train position within the VBF. In this case, the physical train has already moved to VSS21 and the VBF still sees the train in VSS12. Note that this is a very common situation as trains usually only send its position cyclically (e.g., each 6 s). Otherwise, this state can be seen as the situation where Train1 has already sent its position report, but the VBF has not yet received it due to the delays of the communication interface.

In summary, with the environment model it is possible to trigger all interfaces of the VBF by generating the following inputs:

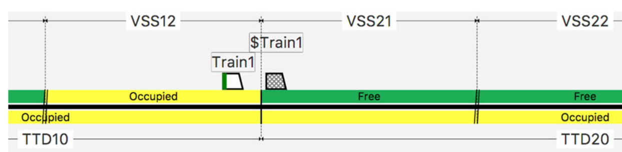

Fig. 3 Environment model: "physical" train position ($Train1) versus train position image in the VBF (Train1)

- Train position reports including train integrity information
- Train registration message (i.e., "Start of Mission" in [1])
- Train deregistration message (i.e., "End of Mission" in [1])
- Train data message (includes the train length)
- TTD occupation status
- Movement Authorities (MA) for trains

The environment model can make use of different tracks. For example, we used the track snippet from the HL3 specification to validate its scenarios and used the real track for on-site execution and to define a test plan for on-site execution.

While animating the scenarios of the HL3 specification, we detected more issues.[9] One issue, which is easy to understand but hard to find without tool support, is the following: in scenario 4 (Start of Mission/End of Mission) at step 8, it is stated that all VSS of TTD 20 go to "unknown" because the disconnect propagation timer of VSS 22 has expired. This is wrong because after the deregistration (end of mission) of the train in step 7, the train will be immediately treated as a ghost train and the corresponding transition #1A will apply. The result for the remaining VSSs of TTD20 is the same but at a different point in time; the VSSs go directly to "unknown" and not just after the disconnect propagation timer (of VSS22) has expired.

As an aside, we think that transition #1A is erroneous, too: there should be an "and" instead of the "or" in *"(no FS MA is issued or no train is located on this TTD)"*. Otherwise, a connected train (with a FS MA) which physically enters a free TTD would always be treated as a ghost train because the TTD occupation usually arrives before a new train position report. In this case, the second condition *"no train is located on this TTD"* would be fulfilled which would allow applying transition #1A. Moreover, transition #1A contains a further ambiguity: *"no FS MA is issued"* could be interpreted as *"no FS MA is issued on this VSS"* or as "no FS MA is issued on this TTD". We opted for the second interpretation based on the scenarios we investigated.

Replaying the other scenarios also revealed a number of inconsistencies compared to the state machine. For example in scenario 9 at step 5, the state of VSS31 went to "unknown" instead of "free" during the animation. According to the scenario description, transition #9B should have been applied, but its second condition "the 'shadow train timer A' of the TTD was not expired at the moment of the time stamp in the position report" was not satisfied. That is because the condition refers to the "shadow train timer A" of the wrong TTD. It should refer to "shadow train timer A" of the TTD in REAR. By investigating step 5 of this scenario more thoroughly we saw some similarities of the transitions #9B and #11A as they are both executed in the same state machine run (caused by the rear-end update) and both refer to the shadow train timer A. Of course they produce different state machine transitions ("ambiguous" to "occupied" vs. "ambiguous" to "free") so their first two conditions differ, but both transitions are used to exclude the shadow train risk. From a logical point of view we can infer the following implication (**):

> If transition #9B is applied to a VSS, then transition #11A should be applied to the next VSS (if we neglect the fact that #9B could also be applied to the next VSS).

The problem with transition #11A is that it contains an additional third condition which is not part of transition #9B: "the reported min-safe-rear-end of this train is within the distance that can be covered at the reported speed within the "shadow train timer A" from the TTD limit". If this condition is not satisfied, our logical implication (**) stated above will not hold. In the scenario, this violation of the implication would lead to the case that VSS32 would not go to "occupied" but to "ambiguous" while VSS31 would still go to "free". While "ambiguous" means that there is a shadow train risk, "free" means that there is no shadow train risk and the VSS is released for following trains. We eliminated this inconsistency by adding the third condition of transition #11A to transition #9B because it provides an additional check against shadow trains.

Besides that, we detected an inaccuracy in this check. The condition says that the min-safe-rear-end of the train must be "within the distance that can be covered [...] from the TTD limit". Thus, our interpretation was that the min-safe-rear-end must be beyond the TTD border but in some situations the min-safe-rear-end can be in rear of the TTD border due to a large confidence interval. In these cases the transition #11A (as well as #9B) should also be applied. In theory, this issue could have been found by the model checker if we had varied the value of the safe train length (reported by the train) in the environment model. However, we only observed the problem in later simulation runs with a train simulator used by Thales but we investigated and solved the problem by replaying the simulation run in the animator.

Besides the validation of the scenarios, the environment model permitted us to investigate system-level safety properties. For example, the system state shown in Fig. 4 should

[9] Overall we detected more than 30 issues which we reported to authors of the HL3 specification.

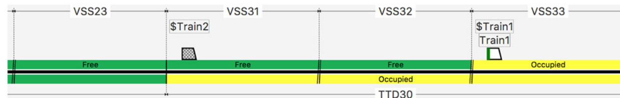

Fig. 4 Invalid system state: non-connected train ($Train2) is located on a VSS with state "free"

never occur. Here, a physical train ($Train2), which is not connected, is located on a VSS which is seen as "free". The threat in this situation is that another train (not displayed in the figure) in rear of the non-connected train could receive a movement authority (FS MA) for VSS31 and VSS32. We were able produce a scenario which finally led to this state: Assume we have the start situation as displayed in Fig. 5. $Train2 has a FS MA up to the end of VSS31. Due to a communication failure $Train2 will not report its position by a train position report while moving to VSS 31. Note that all VSS on TTD2 in Fig. 6 will not change their state even if TTD2 becomes occupied because train2 is within its MA and is not yet a ghost train (note, that we have already integrated the fix for transition #1A as stated above). If now $Train2 is on VSS31 and its mute timer expires, only VSS13 (last reported train location) will go to unknown (according to 4.2.1.3 of HL3 specification: "As the train with a communication failure could have used its MA completely, all the VSS in advance of the last train location which are part of the MA sent to that train shall also be set to 'unknown' immediately, **but only up to the first TTD which is free."**, here the textual description is inconsistent with transition #1B). Even if the ghost train propagation timer expires (which is started by the expired mute timer), the VSSs of TTD3 will not be affected as can be seen in Fig. 7 (according to 4.2.1.4 of HL3 specification, here the textual description is inconsistent with transition #1D).

6.2 Tooling enhancements

The chosen technology influenced the modelling to the extent, that we developed a constructive version of the specification. The modelling activities also influenced the tooling. Below we elaborate on some extensions.

6.2.1 Replaying recorded runs with ProB

We added a feature in ProB to write all executed B operations in a log file (see Fig. 2). This enabled the VBF to log simulations runs (with On-Board-Unit simulators) as well as demonstration runs (with real trains). These runs could then be replayed later in the ProB animator. This was vital, as it allowed us to analyse defects without inspecting (huge) RBC, IXL and Java log files. Log replay was also used to define timer values of the HL3 specification.

6.2.2 HTML export

We implemented a feature to export ProB's history as an HTML file, showing the individual events and a graphical representation of all the steps. This was useful to examine a scenario offline (on paper, by external persons), without having to run the animator.

6.3 Further validation activities

6.3.1 Constraint-based analysis

In the initial stages of the development of the formal B model, we used the ProB constraint solver to look for loops in the VSS state machine. The constraint solver detected issues in the initial versions of our models, where a series of transitions could fire indefinitely, from the same starting position without any outside events.

For this, we generated a new high-level analysis model, which includes the B machine encoding the VSS state transitions (see Sect. 5.2). This high-level model contained 38 combined events such as the following one, capturing a cycle over transitions 5A and 10B (from unknown to ambiguous back to unknown), using the sequential composition operator ";".[10] If the combined event is enabled, it can thus be applied infinitely often.

```
loop_5A_10B(vss) = PRE vss : VSS THEN
   VSS_Unknown_Ambiguous_5A(vss);
   VSS_Ambiguous_To_Unknown_10B(vss)
END;
```

We then used ProB to check feasibility of these events, i.e., is there a state that satisfies the invariant and enables the event under consideration. This analysis produced counter example states for early, faulty versions of our model. In the current version of the model, no such loops are detected by the constraint solver.

6.3.2 Model checking

Another early validation activity was to check commutativity of the VSS state transition wrt the order in which events arrived at the VBF. For example, when a train leaves a TTD, the VBF could first receive a position report outside of a TTD or the updated TTD status information. The TTD update also consists of two events: freeing the previous TTD and occupying the following TTD. There again, we wanted to check whether our model was commutative wrt the order in which these updates were received.

This validation was done using focussed model checking, by hard-coding a scenario (e.g., a scenario from HL3

[10] As such this is not an event allowed by pure Event-Bor RODIN.

Fig. 5 Invalid system state: start situation

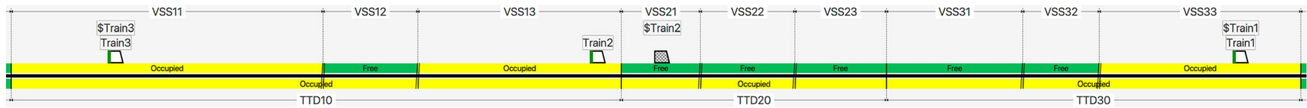

Fig. 6 Invalid system state: physical $Train2 is moving without sending a position report

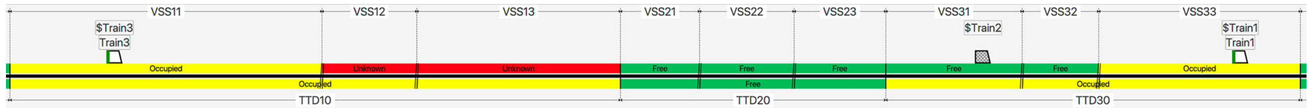

Fig. 7 Invalid system state: VSSs of TTD3 are not affected by the disconnecting Train2

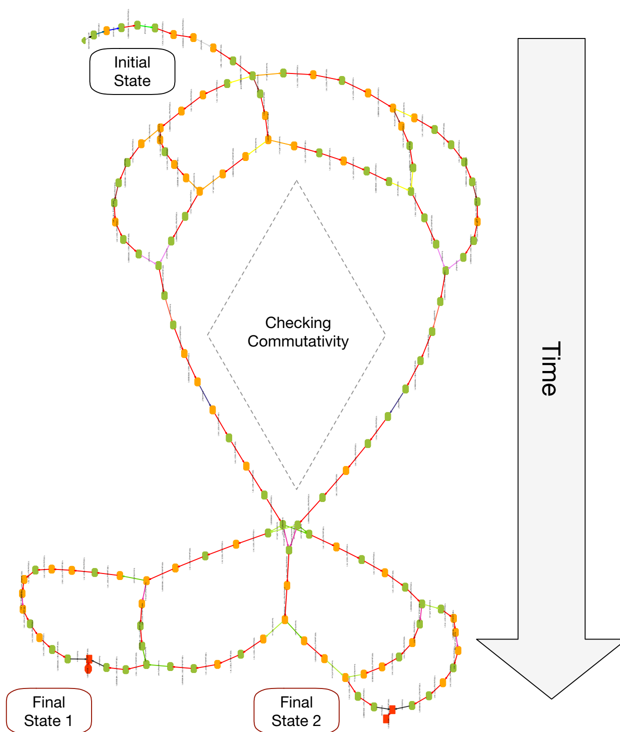

Fig. 8 Checking commutativity of the VSS state machine using model checking

specification), but allowing different interleavings to occur for the above-mentioned update events. This produced small state spaces such as the one in Fig. 8. The initial state of the scenario is at the top, and two possible outcomes are at the bottom of the figure. As one can see, this model was **not** commutative, due to the presence of two distinct possible final states, depending on the order of the events. The validation was useful in understanding HL3 specification and improving our model.

6.4 Visualisation

One requirement for the actual on-site field demonstration was to provide a visualisation for checking the correct functioning of the VBF. Additionally, our experience has shown [22–24] that a visualisation combined with an interactive animator can be especially useful in early stages of the development such as the modelling and analysis stage.

Thus, our intention was to develop one visualisation that could be used in the early stages and during the field demonstration. As a consequence, the visualisation was developed as a separate software component with clearly defined interfaces for it to be integrated both into the ProB-Animator and the final VBF product. In both cases, the state information is extracted from the same (core) model. The only difference is that within the ProB-Animator the model is interactively controlled by a human (typically a domain expert using ProB) and in the final VBF software, the model is controlled via the real interfaces of the VBF.

Having the visualisation in the early stages of the project provided the following benefits:

– We quickly spotted mistakes in the specification and the model.
– We used the visualisation to communicate the model within our team and to the domain experts.
– We were able to replay the scenarios in the HL3 specification and detected inconsistencies between them and the state machine description.
– The visualisation enabled us to let a domain expert act as a tester by interactively inspecting the model (Figs. 9, 10).

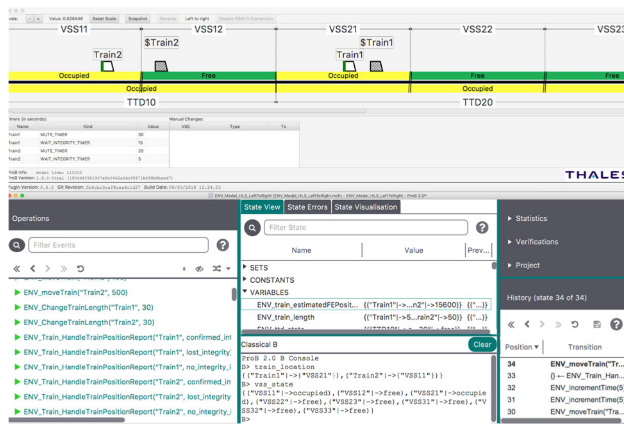

Fig. 9 Screenshot of the visualisation running as a ProB-Animator plugin

Fig. 10 Formal B model being executed in real time with Thales train simulators

6.5 On-site field tests

Building upon the Thales domain knowledge, the formal B model was developed from July until the end of October (including the embedding application), with fine-tuning performed afterwards. A first integration with the Thales RBC was carried out in the beginning of November. The field demonstrations were carried out in November and December 2017.

Later another field test was conducted together with the Deutsche Bahn in September 2018, resulting in a video[11] for the Innotrans trade fair.

7 Other observations

7.1 Roles of ProB

ProB had two different roles in our project. Its first role was the execution engine for our B model. From the formal methods perspective, it is interesting to note that the B model can be used to control simulated and real trains in real time. Moreover, despite the complexity of the model and the fact that all states and transitions were stored, no problems with ProB occurred at runtime, performance and memory consumption were no issues.[12] In addition, the ProB Java API turned out to be a flexible way to link a formal model to external data sources or components.

In its second, more common role, ProB was the central tool in the validation process of the model and specification. Animation combined with visualisation were crucial for the success of the project, in particular to replay and validate the scenarios of the HL3 specification. We think this approach, of using animation and custom visualisations at every stage of development—especially the early ones—should be more widely used for safety critical (e.g., SIL 4) projects in industry. For example, the specification engineer can take over some work of the testing team as he or she is able to interactively derive test cases from the model,[13] which are much more precise and consistent compared to the description of the scenarios contained in the HL3 specification.

7.2 B versus event-B

A good summary of the differences between B and Event-B can be found in [25]. From a scientific perspective it is interesting to analyse why we did not use Event-B as provided by the Rodin platform [9], as opposed to, e.g., [26] or four case study contributions at ABZ'18 (see Sect. 8).

First, our B model had to be run with real track data and real train data. As such we needed data structures such as strings and string manipulation. These are not available in Rodin. Also, to process train location reports and read the topology XML files, we needed to write intricate conversion procedures. These are easier to write in classical B syntax using ProB's extensions such as IF-THEN-ELSE or LET for expressions (see [17]), as well as string manipulation functions. Also, parts of our model used WHILE loops, e.g., to repeatedly apply changes to the VSSs (see Sect. 5.2) or for computing topology information (during VBF startup). This would have been more involved in Event-B, requiring the specification of an event scheduler or using the code generation rules from Chapter 15 of [19].

Concerning systems modelling, it turns out that the classical B machine composition primitives (like SEES and INCLUDES) are sometimes sufficient to decompose a sys-

[11] https://www.youtube.com/watch?v=K6mS6akRmvA.

[12] For example, in one 6-min run ProB's response time was—with one exception—between 0.02 and 0.14 s per event. One event required 0.38, one 0.31 and one 0.27 s, probably due to garbage collection being triggered.

[13] Note that we talk here about product and system-level tests and not just unit tests.

tem into components. Synchronisation by events can then be achieved by a main machine which calls the operations of the included machines in an appropriate manner. The following sketches how a system machine can include two components and then synchronise events between these two components.

```
MACHINE ComponentA
 OPERATIONS EventA
 ...
END
MACHINE ComponentB
 OPERATIONS EventB
 ...
END
MACHINE System
 INCLUDES ComponentA, ComponentB
 ...
 OPERATIONS
   SystemUpdate = ... EventA || EventB ....
END
```

Finally, from a pragmatic point of view, it is still easier to work as a team on models with textual representation as offered by Atelier-B rather than the XML database of Rodin. The textual representations can be more easily shared and versioned (e.g., using git) and refactored using standard text editors. The use of directories to structure our development (see Sect. 4) was also helpful to manage model evolution.

8 Comparison

Four other case studies presented at ABZ'18 used Event-B:

1. The Event-B model [21] by Ammar et al. focussed on proof and trying to establish determinacy of the VSS state machine (which is related to our commutativity analysis in Sect. 6.3.2). The model used four refinements to prove a safety property. ProB was used to validate the model. In contrast to our model, the operations acted on all VSSs in one go; we have one VSS as parameter for each transition (see Sect. 5.2). The model has 10 constants and 26 variables.
2. The article [27] is a companion paper to [21] and focusses on requirements modelling using SysML and KAOS. This issue is orthogonal to our paper.
3. The model [28] by Abrial focusses on proof on a simplified version of the HL3 specification. It is a "green field" development of the concept and does not use refinement. The model has 5 constants and 15 variables.
4. [29] used iUML translated to Event-B; has a refinement strategy with 5 refinements. The last level has 13 variables and 9 constants and can be animated with ProB.

Note that our model is the largest with 45 constants and 33 variables, but we did not conduct any proof. It is interesting to note that the above Event-B developments all used Rodin, while we did not (see Sect. 7.2). We pursued a bottom-up approach, starting from VSSs and TTDs up to modelling trains and VSS update cycles. Some approaches above started top-down, from an obviously safe system and adding details via refinement. This is certainly a viable approach for proving the safety of a system. However, we ourselves do not yet have enough understanding of HL3 specification to understand why it is safe and how a proof and refinement strategy should look like. Developing a system-level proof of HL3 specification is worthy of another research project, and can get inspiration from successful use of Event-B for similar demonstrations for the Flushing line in New York [14] or the Octys line in Paris [15].

Two other solutions were presented at ABZ'18, one using Spin [30] and one using Alloy/Electrum [31]. The latter is interesting as it uses Alloy's magic layout feature to obtain a visual representation of the state of the model. Visualisation was extremely useful for our model, and helped to communicate with domain experts. It is interesting to note that [30] mentioned that visualisation was one missing feature of Spin.

An important factor is that our model is a complete, executable realisation of HL3 specification. The differences in the model and the methodology partly stems from the fact that we had been given a clear goal, outside of the case study: to get a complete VBF implementation ready by December 2018 for on-site field testing. On the one hand, we thus had access to a larger team, with domain experts, that helped us unravel the obscure parts of HL3 specification. On the other hand, we had a given budget and tight deadline, leaving little room for activities such as proof and refinement.

9 Conclusions

As we implemented a full executable application based on the HL3 case study rather than being concerned with proving the model, we gained additional insights in the usefulness of tool support. For example, logging all B events (see Fig. 2) during the field tests proved to be vital: due to communication between several components, a single run was inherently non-deterministic. Full logs of events and parameters allows replaying and analysing the trace later on, e.g., in fully controlled environments like an animator on a machine independent of the field-test system.

Another useful feature would be a test suite that is hooked up to an animator. Much time was spent on replaying the scenarios from the specification after changing the model. It would be greatly appreciated if specified scenarios could be replayed automatically on change, and, if applicable, differences in encountered states are immediately reported. Therefore, in the future, we plan to put more focus on tool-

ing for scenario replay and B model regression tests (see also [32]).

From the project, we can conclude that formal models can be useful and cost-effective for demonstrators. Animation with forward/backward stepping and visualisation were extremely useful in the development process. We were able to develop a complete formalisation of the HL3 specification: the B formal model can now serve as an *executable reference specification*, for understanding the HL3 principles, for deriving test cases from it or possibly to generate code using Atelier-B.

Acknowledgements Open Access funding provided by Projekt DEAL. We thank Jens Bendisposto, David Geleßus, Christoph Heinzen, Antonia Pütz, Yumiko Takahashi, Fabian Vu and Michelle Werth for all the work that went into the ProB Java API and the new ProB-Animator UI. We thank Mirko Aigner, Stefano Allrath, Burkhard Börner, Joachim Jost, Editha Nentzl, Sebastian Neuhauß, Michael Schilling, Wilfried Seibt, Tom Seidel and Tino Wegner from Thales as well as the staff from ClearSy for their work and support on the demonstrator. Moreover, we are thankful to the authors of the HL3 specification and the reviewers of ABZ and STTT for their very useful feedback and patience.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Hybrid ERTMS/ETCS level 3. Principles Ref: 16E042, Version: 1A, EEIG ERTMS Users Group, 123-133 Rue Froissart, 1040 Brussels, Belgium (2017)
2. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
3. Jiang, Z., Pajic, M., Moarref, S., Alur, R., Mangharam, R.: Modeling and verification of a dual chamber implantable pacemaker. In: Proceedings TACAS'2012, Volume 7214 of LNCS, pp. 188–203. Springer, Berlin (2012)
4. Hansen, D., Leuschel, M., Schneider, D., Krings, S., Körner, P., Naulin, T., Nayeri, N., Skowron, F.: Using a formal B model at runtime in a demonstration of the ETCS hybrid level 3 concept with real trains. In: Proceedings ABZ'2018, Volume 10817 of LNCS, pp. 292–306. Springer, Berlin (2018)
5. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
6. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, London (1992)
7. Dollé, D., Essamé, D.: B dans le transport ferroviaire. L'expérience de Siemens Transportation Systems. Tech. Sci. Inform. **22**(1), 11–32 (2003)
8. Essamé, D., Dollé, D.: B in large scale projects: the Canarsie line CBTC experience. In: Proceedings B'2007, Volume 4355 of LNCS, pp. 252–254. Springer, Berlin (2007)
9. Abrial, J.-R., Butler, M., Hallerstede, S.: An open extensible tool environment for Event-B. In: Proceedings ICFEM'2006, Volume 4260 of LNCS, pp. 588–605. Springer, Berlin (2006)
10. Essamé, D., Dollé, D.: B in large-scale projects: the Canarsie line CBTC experience. In: Proceedings B'2007, Volume 4355 of LNCS, pp. 252–254. Springer, Berlin (2007)
11. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale B models with ProB. Form. Asp. Comput. **23**(6), 683–709 (2011)
12. Lecomte, T., Burdy, L., Leuschel, M.: Formally checking large data sets in the railways. CoRR, arXiv:1210.6815 (2012)
13. Sabatier, D., Burdy, L., Requet, A., Guéry, J.: Formal proofs for the NYCT line 7 (flushing) modernization project. In: Proceedings ABZ'2012, pp. 369–372 (2012)
14. Sabatier, D.: Using formal proof and B method at system level for industrial projects. In: Proceedings RSSRail'2016, Volume 9707 of LNCS, pp. 20–31. Springer, Berlin (2016)
15. Comptier, M., Déharbe, D., Perez, J.M., Mussat, L., Thibaut, P., Sabatier, D.: Safety analysis of a CBTC system: a rigorous approach with Event-B. In: Proceedings RSSRail'2017, Volume 10598 of LNCS, pp. 148–159. Springer, Berlin (2017)
16. Leuschel, M., Butler, M.J.: ProB: a model checker for B. In: Proceedings FME'2003, Volume 2805 of LNCS, pp. 855–874. Springer, Berlin (2003)
17. Hansen, D., Schneider, D., Leuschel, M.: Using B and ProB for data validation projects. In: Proceedings ABZ'2016, Volume 9675 of LNCS, pp. 167–182. Springer, Berlin (2016)
18. Schneider, D., Leuschel, M., Witt, T.: Model-based problem solving for university timetable validation and improvement. In: Proceedings FM'2015, Volume 9109 of LNCS, pp. 487–495. Springer, Berlin (2015)
19. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
20. ERTMS/ETCS—Baseline 3. System Requirements Specification Ref: SUBSET-026-3, Issue: 3.0.0, EEIG ERTMS Users Group, 123-133 Rue Froissart, 1040 Brussels, Belgium (2008)
21. Mammar, A., Frappier, M., Fotso, S.J.T., Laleau, R.: An Event-B model of the hybrid ERTMS/ETCS level 3 standard. In: Proceedings ABZ'2018, pp. 353–366 (2018)
22. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B models with B-Motion Studio. In: Proceedings FMICS'2009, Volume 5825 of LNCS, pp. 202–204. Springer, Berlin (2009)
23. Ladenberger, L.: Rapid Creation of Interactive Formal Prototypes for Validating Safety-Critical Systems. PhD thesis, University of Düsseldorf, Germany (2017)
24. Ladenberger, L., Hansen, D., Wiegard, H., Bendisposto, J., Leuschel, M.: Validation of the ABZ landing gear system using ProB. Int. J. Softw. Tools Technol. Transf. **19**, 187–203 (2017)
25. Abrial, J.-R.: On B and Event-B: principles, success and challenges. In: Proceedings ABZ'2018, Volume 10817 of LNCS, pp. 31–35. Springer, Berlin (2018)
26. Reichl, K., Fischer, T., Tummeltshammer, P.: Using formal methods for verification and validation in railway. In: Proceedings TAP'2016, Volume 9762 of LNCS, pp. 3–13. Springer, Berlin (2016)
27. Fotso, S.J.T., Frappier, M., Laleau, R., Mammar, A.: Modeling the hybrid ERTMS/ETCS level 3 standard using a formal requirements engineering approach. In: Proceedings ABZ'2018, Volume 10817 of LNCS, pp. 262–276. Springer, Berlin (2018)
28. Abrial, J.-R.: The ABZ-2018 case study with Event-B. In: Proceedings ABZ'2018, Volume 10817 of LNCS, pp. 322–337. Springer, Berlin (2018)

29. Dghaym, D., Poppleton, M., Snook, C.F.: Diagram-led formal modelling using iUML-B for hybrid ERTMS Level 3. In: Proceedings ABZ'2018, Volume 10817 of LNCS, pp. 338–352. Springer, Berlin (2018)
30. Arcaini, P., Jezek, P., Kofron, J.: Modelling the hybrid ERTMS/ETCS level 3 case study in Spin. In: Proceedings ABZ'2018, Volume 10817 of LNCS, pp. 277–291. Springer, Berlin (2018)
31. Cunha, A., Macedo, N.: Validating the hybrid ERTMS/ETCS Level 3 concept with electrum. In: Proceedings ABZ'2018, Volume 10817 of LNCS, pp. 307–321. Springer, Berlin (2018)
32. Snook, C.F., Hoang, T.S., Dghaym, D., Butler, M.J., Fischer, T., Schlick, R., Wang, K.: Behaviour-driven formal model development. In: Proceedings ICFEM'2018, pp. 21–36 (2018)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.