Using a Formal B Model at Runtime in a Demonstration of the ETCS Hybrid Level 3 Concept with Real Trains

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

[1] Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, Düsseldorf, Germany
{hansen,leuschel,schneider,krings,korner}@cs.uni-duesseldorf.de
[2] Thales Deutschland GmbH, Berlin, Germany
{thomas.naulin,nader.nayeri,frank.skowron}@thalesgroup.com

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 (HL3) introduces Virtual Sub-Sections (VSS) as sub-divisions of classical track sections with Track-side Train Detection (TTD). 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 VSSs according to the HL3 concept using the train position reports, train integrity information, and the TTD occupation states. From the perspective of the RBC, the VBF behaves as an Interlocking (IXL) that transmits all signal aspects for virtual signals introduced for each VSS 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 · ETCS

1 Introduction and Requirements

The 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. The main difference between the HL3 concept and a solution without any trackside train detection (pure Level 3) is that not all trains need to be equipped with an ETCS on-board unit and a TIMS (Train integrity monitoring system). In addition, the information from the underlying trackside train detection system can be used as fall back to, e.g., handle degraded situations and to improve the performance.

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.

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 Sub-Sections (**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. For example, in Fig. 1 at the bottom you can see that we have two areas each with a trackside detection device (realised by axle counters or track circuits). 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.

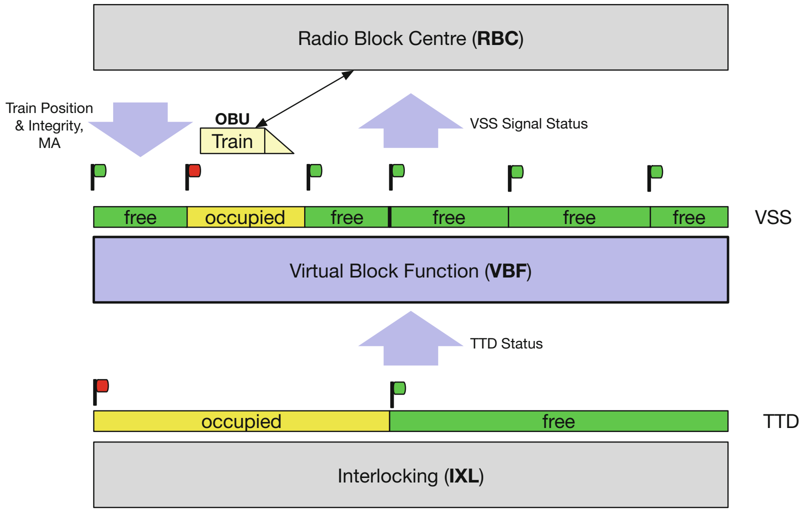


Fig. 1. The role of the VBF (Virtual Block Function)

In the following sections, we will report on our experience building a software product for the VBF based on a formal B model. In Sect. 2 we outline our tasks and early design decisions. Section 3 provides an overview of the formal B model and the modelling challenges, along with some ambiguities and inconsistencies we found in the HL3 specification. Section 4 describes the architecture of VBF software which embeds the B model. Visualisation was important in our project and we discuss it Sect. 5. We conclude with discussion about practical results and insights gained in Sect. 6.

2 Project Constraints and Design Decisions

Due to the strict deadline 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** [2] 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.[1]
- The VBF had to be developed from scratch as an add-on for the RBC, which was to mimic an interlocking and transmits 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 paper:

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 be 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., [3–9]) 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.

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

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 ProRail, 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 [10] as the execution engine in different software products [11,12]. 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.

3 The Formal B Model

Below, we present some relevant aspects of our B model along with some source code snippets. Due to space limitations we cannot cover all interesting aspects, such as the modelling of timers and time.

3.1 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., Chap. 14 of "Modeling in Event-B" [13]).

However, for pragmatic reasons, we did not use Event-B [13] but rather classical B [2] for modelling the VBF. For example, we have modelled the VSSs, TTDs and trains as classical B strings. For simulation and execution purposes, we had to read topology and configuration data from XML files. The conversion of the XML file into B data structures for the VBF model is also done in classical B using records and strings.[2] Finally, we have used other features, such as machine composition and operation calls (see Sect. 3.2), not readily available in Event-B.

Below, we try to give a flavour of our modelling by showing some derived data structures for the track topology.

```
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 two states
& next_ttd : TTD +-> TTD
& last_vss: TTD --> VSS
```

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

```
& /*@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.[3] However, the direction of the track can be toggled, since the conversion of the XML data is parameterised. Observe that we allow the IF-THEN-ELSE to be applied to expressions and use an external B function (see Sect. 6.3 in [11]) 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
     } )
```

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,
                               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
```

[3] 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.

```
  registeredTrains : POW(STRING) &
& train_reportedTrainIntegrity
                   : registeredTrains --> REPORTED_TRAIN_INTEGRITY
& train_integrity  : registeredTrains --> INTERNAL_TRAIN_INTEGRITY
...
```

According to the ERTMS/ETCS specifications [14], 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. Within the VBF, we only need to distinguish between three, which are represented by the enumerated set REPORTED_TRAIN_INTEGRITY. The surjective function TRAIN_INTEGRITY_MAPPING defines the respective mapping.

Moreover, the HL3 specification [1, Sect. 3.5] defines a further integrity state by using the terms "integer" and "not integer" which is represented by the enumerated set INTERNAL_TRAIN_INTEGRITY.[4] Yet, an unambiguous mapping from the reported train integrity to the internal train is missing in the HL3 specification [1]. 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"*

Both conditions are part of the transitions #7B and #8A [1, Sect. 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, integrityStatus) =
    PRE integrityStatus : REPORTED_TRAIN_INTEGRITY
    THEN
      train_reportedTrainIntegrity(train) := integrityStatus ||
      IF integrityStatus=lost_integrity
      THEN  train_integrity(train) := not_integer
      ELSIF integrityStatus = 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
```

[4] The term "internal" refers to the internal state of the VBF.

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, Sect. 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, Sect. 3.3.3.1]. Here, the term "integer train" is used which corresponds to the internal train integrity of our model. However, in Sect. 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, Sect. 3.3.3.4]. Thus, Sect. 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".

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)
  & /*@label The train location must not have any gaps */
    !loc.(loc: ran(train_location)
      => #s.(s : iseq(loc)
            & !i,ii.(i : 1..size(s-1) => s(i) |-> s(i + 1) : next_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` is very convenient. Alternatively, we could have used a relation but we prefer functions over relations except for the `next_vss` constant which is frequently inverted in our model. The order of the VSS is not incorporated into the location definition as this information is already contained in the `next_vss` constant. 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 this constant.

While the modelling of the train location data structures was relatively straightforward, the updates to this variable are, in our opinion, the most under-specified 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, Sect. 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. (The other alternative, forced us to adapt several transitions in order to be able to replay all scenarios of the HL3 specification.)

3.2 State Machine Transitions and Priorities

Below, we show the B translation of the state machine transition (#9A) of the HL3 specification.

```
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
```

The reason for separating out the guards into DEFINITIONS (in a separate file) is to encode the priorities of the HL3 specification. 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. The IF-THEN-ELSE ensures that the priorities of the transitions are respected, e.g., that transition `2A` has priority over `3`. A return variable `out` stores the exact VSS transition taken for debugging and analysis.

```
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`.

3.3 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 (detection of inconsistencies).

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. Figure 2 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.

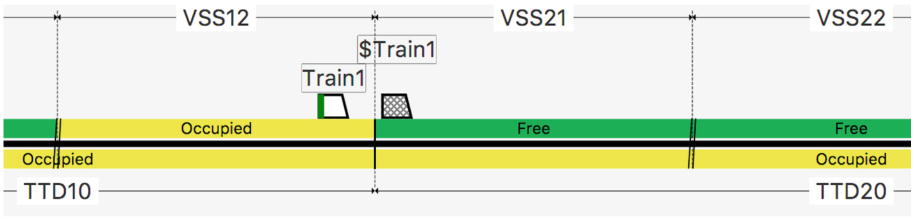


Fig. 2. Environment Model: "physical" train position ($Train1) vs. train position image in the VBF (Train1)

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

– Train position reports including train integrity information
– Train registration message
– Train deregistration message
– 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 onsite execution and to define a test plan for onsite execution.

While animating the scenarios of the HL3 specification, we detected more issues.[5] 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 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.

Besides the validation of the scenarios, the environment model permitted us to specify system level invariants. For example, the system state shown in Fig. 3 should 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 caused by an invalid stopping criterion for the ghost train propagation.[6]

Replaying Recorded Runs with ProB. Simulations runs (with On-Board-Unit simulators) as well as demonstration runs (with real trains) were logged by the VBF and could be replayed in the 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.

[5] Overall we detected more than 30 issues which we reported to authors of the HL3 specification.
[6] The scenario is too complex to be presented in this paper.

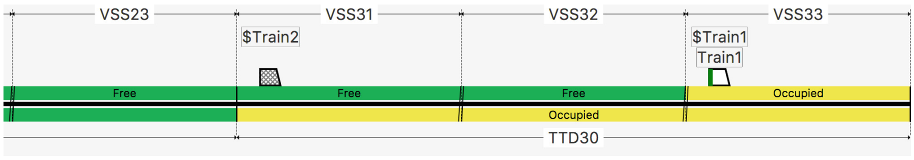


Fig. 3. Invalid system state: Non-connected train ($Train2) is located on a VSS with state "free".

4 Architecture

The VBF model described above is part of a larger application developed to conduct the field demonstration. The application embeds the VBF model using the ProB Java API [15] (often referred to as ProB2) and manages all the model's interactions with the outside world. The Java API exposes all of ProB's animation and model checking features to programs running on the Java Virtual Machine. This approach has been successfully used in several applications that use B models at runtime [12] and is the basis for a new ProB UI that is currently being developed.

The responsibilities of the application are: firstly, to interact with external input sources such as the RBC and others that provide information about the current state of the track, of physical and of simulated trains, etc. Secondly, to process these inputs and forward them to the model. And lastly, to act on the newly computed state of the model to update the visualisation and send updates to the RBC.

Figure 4 provides an overview of the application's architecture. The external inputs are provided via a variety of inputs, such as UDP packages, XML-RPC calls, plain files, etc. These inputs represent train information from the RBC as well as TTD information from real and simulated trains. These events are

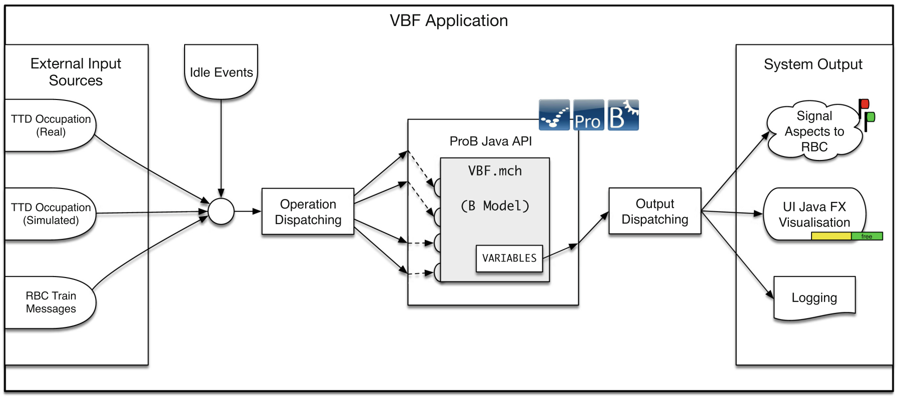


Fig. 4. Application architecture

received by the application, normalised and dispatched to the model. In case there are no external events, the application will, after a given delay, begin sending idle events to the model in specific intervals until it receives new external events. These events are used to update the timers in the model and compute an updated system state even in the absence of external events. Each type of input event is dispatched to a corresponding operation of the B machine by executing one guided animation step and computing a new state of the model. From each new model state computed by ProB, we derive an application-level state representation. This representation is based on the state variables of the model. These variables are exposed through the ProB Java API and extracted from the state, mapped to Java structures and used to compute the application's outputs. From this application-level state the signal aspect changes are extracted and sent to the RBC. The state is provided to the visualisation layer to update the track diagram and information tables. Lastly, the delta between two states is logged for debugging purposes.

5 Visualisation

One requirement for the actual onsite field demonstration was to provide a visualisation for checking the correct functioning of the VBF. Additionally, our experience has shown [16–18] 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 separated 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 via an environment model by a user and in the final VBF software, the model is controlled via the real interfaces of the VBF.

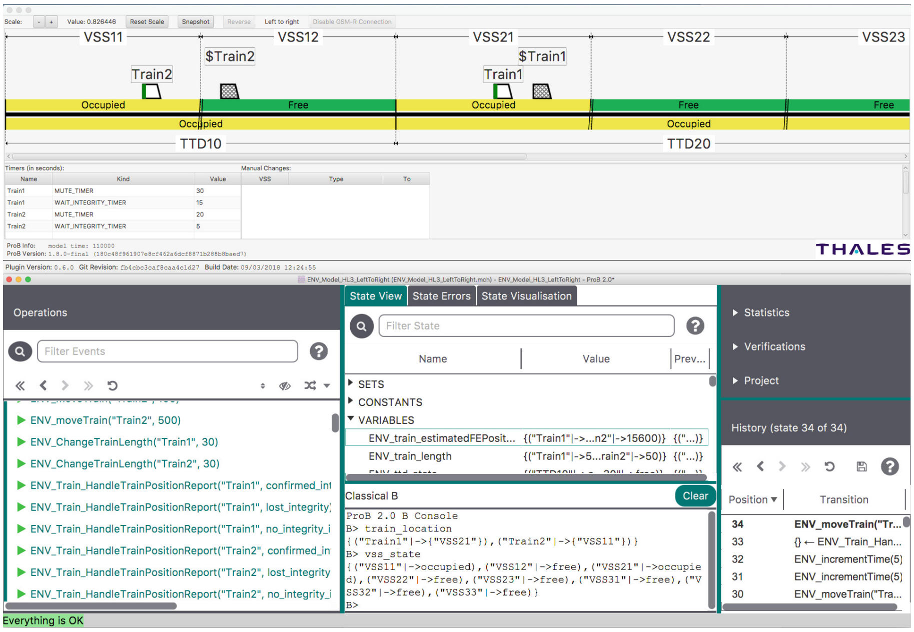
Having the visualisation in the early stages of the project provided the following benefits:

- 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.

For the project, we have also developed a new feature in ProB, namely to export an entire animation trace into an HTML file with one visualisation per state. This feature was useful to send entire animation scenarios to domain experts.

For the main application, we created a custom visualisation using the JavaFX UI framework. The visualisation is linked to the B model's state, and updates itself as soon as a new state is provided. As such, the same visualisation could be used as a plugin in the ProB-Animator during development.

Figure 5 shows a screenshot of the VBF visualisation running as a plugin in the ProB-Animator.



Timers (in seconds):

Name	Kind	Value
Train1	MUTE_TIMER	30
Train1	WAIT_INTEGRITY_TIMER	15
Train2	MUTE_TIMER	20
Train2	WAIT_INTEGRITY_TIMER	5

Manual Changes:

VSS	Type	To

ProB Info: model time: 110000
ProB Version: 1.8.0-final (180c48f961907e8cf462a6dcf8871b288b8baed7)
Plugin Version: 0.6.0 Git Revision: fb4cbc3caf8caa4c1d27 Build Date: 09/03/2018 12:24:55

ENV_Model_HL3_LeftToRight (ENV_Model_HL3_LeftToRight.mch) - ENV_Model_HL3_LeftToRight - ProB 2.0*

ENV_moveTrain("Train2", 500)
ENV_ChangeTrainLength("Train1", 30)
ENV_ChangeTrainLength("Train2", 30)
ENV_Train_HandleTrainPositionReport("Train1", confirmed_int
ENV_Train_HandleTrainPositionReport("Train1", lost_integrity)
ENV_Train_HandleTrainPositionReport("Train1", no_integrity_i
ENV_Train_HandleTrainPositionReport("Train2", confirmed_in
ENV_Train_HandleTrainPositionReport("Train2", lost_integrity
ENV_Train_HandleTrainPositionReport("Train2", no_integrity_i

Name	Value	Prev...
ENV_train_estimatedFEPosit...	{("Train1"|->...n2"|->15600)}	{("...)}
ENV_train_length	{("Train1"|->5...rain2"|->50)}	{("...)}

```
ProB 2.0 B Console
B> train_location
{("Train1"|->{"VSS21"}),("Train2"|->{"VSS11"})}
B> vss_state
{("VSS11"|->occupied),("VSS12"|->free),("VSS21"|->occupie
d),("VSS22"|->free),("VSS23"|->free),("VSS31"|->free),("V
SS32"|->free),("VSS33"|->free)}
B>
```

Position	Transition
34	ENV_moveTrain("Tr...
33	{} ← ENV_Train_Han...
32	ENV_incrementTime(5)
31	ENV_incrementTime(5)
30	ENV_moveTrain("Tra...

Everything is OK

Fig. 5. Screenshot of the visualisation running as a ProB-Animator plugin

6 Practical Results, Discussion, Conclusion

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. The VBF demonstrator was finished on time and on budget, and the demonstration of the HL3 principles using the Thales RBC was successful. The VBF model (without environment) consists of 13 B Machines, 14 definition files and has 45 constants and 28 variables. The required scenarios were demonstrated, with simulated and real trains. Five persons from HHU worked on the VBF demonstrator (two on the formal B model, three on the boundaries and the visualisation). Also, within the project, some ProB extensions were developed.

ProB had two different roles in our project. Its first role was, as described in Sect. 4, 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, no problems with ProB occurred at runtime, performance and memory consumption were no issues.[7] 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 is able to interactively derive test cases from the model[8], which are much more precise and consistent compared to the description of the scenarios contained in the HL3 specification.

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. Jens Bendisposto, David Geleßus, Christoph Hein-zen, 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 Neuhau, 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 for their useful feedback.

References

1. Hybrid ERTMS/ETCS Level 3. Principles Ref: 16E042, Version: 1A, EEIG ERTMS Users Group, 123–133 Rue Froissart, 1040 Brussels, Belgium, 7 2017
2. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
3. Dollé, D., Essamé, D., Falampin, J.: B dans le transport ferroviaire. L'expérience de siemens transportation systems. Tech. Sci. Inform. **22**(1), 11–32 (2003)

[7] For example, in one 6-min run ProB's response time was—with one exception—between 0.03 and 0.14 s per event. One event required 0.31 s, possibly due to garbage collection being triggered.

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

4. Essamé, D., Dollé, D.: B in large-scale projects: the canarsie line CBTC experience. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 252–254. Springer, Heidelberg (2006). https://doi.org/10.1007/11955757_21
5. Leuschel, M., Falampin, J., Fritz, F., Plagge, D.: Automated property verification for large scale B models with ProB. Formal Asp. Comput. **23**(6), 683–709 (2011)
6. Lecomte, T., Burdy, L., Leuschel, M.: Formally Checking Large Data Sets in the Railways. CoRR, abs/1210.6815 (2012)
7. Sabatier, D., Burdy, L., Requet, A., Guéry, J.: Formal proofs for the NYCT Line 7 (flushing) modernization project. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 369–372. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_34
8. Sabatier, D.: Using formal proof and B method at system level for industrial projects. In: Lecomte, T., Pinger, R., Romanovsky, A. (eds.) RSSRail 2016. LNCS, vol. 9707, pp. 20–31. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33951-1_2
9. Comptier, M., Déharbe, D., Perez, J.M., Mussat, L., Pierre, T., Sabatier, D.: Safety analysis of a CBTC system: a rigorous approach with Event-B. In: Fantechi, A., Lecomte, T., Romanovsky, A. (eds.) RSSRail 2017. LNCS, vol. 10598, pp. 148–159. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68499-4_10
10. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
11. Hansen, D., Schneider, D., Leuschel, M.: Using B and ProB for data validation projects. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 167–182. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_10
12. Schneider, D., Leuschel, M., Witt, T.: Model-based problem solving for university timetable validation and improvement. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_30
13. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
14. 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, December 2008
15. Bendisposto, J., Clark, J., Dobrikov, I., Körner, P., Krings, S., Ladenberger, L., Leuschel, M., Plagge, D.: ProB 2.0 Tutorial. In: Proceedings of the 4th Rodin User and Developer Workshop, TUCS Lecture Notes, Turku, June 2013. Turku Centre for Computer Science
16. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B models with B-motion studio. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 202–204. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04570-7_17
17. Ladenberger, L.: Rapid creation of interactive formal prototypes for validating safety-critical systems. Ph.D. thesis, University of Düsseldorf, Germany (2017)
18. Hansen, D., Ladenberger, L., Wiegard, H., Bendisposto, J., Leuschel, M.: Validation of the ABZ landing gear system using ProB. In: Boniol, F., Wiels, V., Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. CCIS, vol. 433, pp. 66–79. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07512-9_5