Modelling and Validating an Automotive System in Classical B and Event-B

Michael Leuschel^(✉), Mareike Mutz, and Michelle Werth

Institut für Informatik, Universität Düsseldorf, Universitätsstr. 1,
40225 Düsseldorf, Germany
`michael.leuschel@hhu.de`

Abstract. We have modelled parts of the ABZ automotive case study using the B-method. For the early phases of modelling we have used the classical B for software, while for proof we have used Event-B and Rodin. It is maybe surprising that classical B's machine inclusion mechanism along with operation calls can be used for modular system modelling. Moreover, for one particular style of modelling, the result can then be translated to superposition refinement with event extension in Event-B. Before conducting the proof, we have validated our models using model checking and animation with visualizations. The graphical visualizations were constructed using a new plugin (VisB) which helped uncover errors and transforms our model into an executable, interactive reference specification which can be examined by users without formal background.

1 Introduction and Background

In this work we have used both classical B and Event-B as modelling languages with PROB and RODIN for tool support. The classical B-method [1] is a formal method rooted in predicate logic, set theory, arithmetic. B arose out of Z, with a focus on tool support and successive refinement to derive provably correct software out of high-level specifications. This initial version of B, supported by the tool ATELIER-B, is now called classical B or also “B for software”.

Event-B was developed to enable systems modelling. It also tried to correct a few issues in classical B, simplifying the language (e.g., making it easier to parse and removing the complex inclusion mechanism) and trying to make refinement proofs easier and more scalable. The main addition though is a more flexible refinement concept targeted at systems modelling rather than software development. The foundations of Event-B are laid out in the book [2]. Event-B is supported by the RODIN platform [3], which we have used for proving. In our development of the case study we made heavy use of the animator and model checker PROB [10] which supports both classical B and Event-B. PROB is also available as a plugin for RODIN.

The goals we set ourselves were as follows:

- Validate the usability of the new visualization plugin VisB for PROB, in particular whether images from the case study description [7] can be reused with little effort.

- Examine whether systems modelling can be conducted in classical B.
- Examine whether a component-based model in classical B can be converted to Event-B for use by the RODIN for proof.

Indeed, in earlier work we have often realised that classical B can also be used for systems modelling and may have some advantages. However, for proving systems, Event-B and RODIN provide a nice platform and simpler¹ proof obligations than classical B. We have modelled a subset of the lighting system, as well as a subset of the adaptive cruise control system. For the latter some insights found during modelling have led to improvements in the case study specification (see Sect. 6). In this article we only show details of the lighting system models, which we have developed using a development methodology described in Sect. 2, which consisted of using classical B during the exploration phase and Event-B for the proving phase.

The contributions of this article thus consist in showing the usefulness of visualisation using a new lightweight SVG-based tool, comparing classical B and Event-B for systems modelling, and illustrating a development methodology. Together with the visualisation, our model provides an executable reference specification for a subset of the specification, which can be used by domain experts without formal methods training. We plan on extending this subset in future.

2 Modelling Methodology and Strategy

The methodology we used was inspired by earlier experience on modeling and validating railway systems (e.g., [4,5]). We divided the modelling into three phases (see Fig. 1):

1. an **exploratory** modelling phase where one tries to understand the specification and domain and identifies important subcomponents.
2. a **synthesis** phase, where the subcomponents are integrated to form ever larger subsystems and the integration is at least partially verified. We could have named this also the *combination* or *structuring* phase.
3. an exhaustive **verification** phase, where the completed system is now formally proven to be safe and functionally correct.

The exploratory phase 1 is dominated by editing and animation. Here it is important that one can quickly change the model, and one wants to use a maximally powerful language. Hence, in our case, we have used classical B with its rich substitution language for this phase. Also, the fact that we have a textual representation (which was versioned in Git) allows for easy editing and collaboration. Animation and graphical visualization are vital to check the presence of desired functionality, but also to check for unexpected behaviour and to gain

¹ E.g., due to the absence of an if-then-else substitution and the use of witnesses the refinement proof obligations are much simpler.

Fig. 1. Machines and events in the three modelling phases

a better understanding of the problem domain. To cite Bryan Cantrill:² “*The visual cortex is unparalleled at detecting patterns.*” and “*The value of visualization is not merely providing answers but especially provoking new questions.*” The outcome of phase 1 is a first decomposition of the system into functional subcomponent models which can be animated.

In phase 2 the model starts to stabilise, but we still experimented with various ways to integrate the components. In a refinement-based approach (e.g., Event-B) one has to decide upon a particular refinement order in which components are integrated, which can be sometimes difficult to find and tedious to change. Hence we again stayed in the classical B paradigm where the powerful machine inclusion features allowed us to experiment with various ways of assembling the entire system (see also Fig. 1). To ensure the proper functioning we start to write safety invariants in this phase. The verification in this phase was dominated by (safety) model checking and trace replay in the animator. The outcome of phase 2 is a functional composition into an overall system which can be animated and where various safety invariants can be checked using model checking.

The last phase is dominated by proof. Here, the RODIN platform’s proof system is a key technology and the simplicity of Event-B’s proof obligations can pay off. In this phase the decomposition of the system has stabilised and it was hence possible to transform the classical B model into a linear refinement hierarchy. This requires translating machine inclusion to machine refinement and operation calls to event extension (see Fig. 1 and Sect. 4). For functional aspects of the system we complemented proof with LTL model checking.

² <https://www.slideshare.net/bcantrill/visualizing-systems-with-statemaps>.

3 Classical B Modelling Details

Initially we had modelled a subset of the cruise control system, which led to a few interesting insights and issues (see also Sect. 6). However, to illustrate our modelling strategy (Sect. 2) and new visualization tool (Sect. 2) we then modelled a subset of the light system. Our models are available at: <https://github.com/hhu-stups/abz2020-models>.

Due to time restrictions we were unable to model the complete case study. We concentrated on the management of the blinkers, i.e., the `blinkLeft` and `blinkRight` actuators as influenced by the vertical position of the Pitman arm (`PitmanArmUpDown` sensor), the hazard warning switch (`hazardWarningSwitchOn` sensor), as well as obviously by timing constraints. The `engineOn` and `keyState` sensor values were also used.

3.1 Components

At the end of the exploratory phase (see Sect. 2) we had separated out the system into three components (see Fig. 2):

1. A small component `Sensors` to manage the state of all sensors of the system and their updates.
2. A component `BlinkLamps` for managing the blinking cycle and the actuators `blinkLeft` and `blinkRight`, based on a logical state variable stipulating which blinkers are active and how often blinking should be repeated.
3. The `PitmanController` composed component which reacts to changes in the sensors and sets the logical state variable of the `BlinkLamps` components.
4. A `GenericTimer` component which manages current time evolution and manages hard deadlines. There are actually two versions of this component, one for animation or simulation and one for finite state model checking.
5. The main `PitmanController_TIME` system, which adds treatment of real time issues to the `PitmanController` model.

3.2 Blink Lamps Controller

The purpose of the `BlinkLamps` component is to separate the management of the blink lamps from a logical state of the system. This logical state is modelled by a variable `active_blinkers` which is a subset of `{left_blink, right_blink}`, i.e., it has four possible logical states (from all blinkers off to all blinkers on). This model does not worry about real time and deadlines yet; it manages various lamp states using an internal state variable `onCycle` but, e.g., does not stipulate how long lamps should remain on or off and does not have a variable representing time at all. The model also manages a `remaining_blinks` variable, which stipulates how often the blinker still has to run independently (used later to implement tip blinking). A special value denotes unlimited continuous blinking. A small part of the model is shown in Listing 1.1. Safety invariants ensure the consistency

Fig. 2. Hierarchy of final classical B model with variables

between the logical variables `active_blinkers` and `remaining_blinks` with the blink lamp actuators. Early versions of our model had several errors in that respect, e.g., when events such that turning an engine on or off changed the system's state without consistently updating the actuators.

```

1 MACHINE BlinkLamps ...
2 DEFINITIONS
3   blinkersOff == (blinkLeft=lamp_off & blinkRight=lamp_off)
4 INVARIANT
5   active_blinkers <: BLINK_DIRECTION &
6   remaining_blinks : BLINK_CYCLE_COUNTER &
7   blinkLeft: LAMP_STATUS & blinkRight : LAMP_STATUS & onCycle: BOOL &
8   /*@label "SAF1" */
9   ((remaining_blinks=0 & blinkersOff) <=> active_blinkers={}) & ...
10  /*@label "SAF6" */
11  (onCycle=TRUE & active_blinkers/={} => not(blinkersOff))
12 OPERATIONS
13   SET_BlinkersOn(direction,rem) = PRE direction: BLINK_DIRECTION & rem:
14   BLINK_CYCLE_COUNTER & rem /= 0
15   THEN
16     active_blinkers := {direction} || remaining_blinks := rem ||
17     IF direction=right_blink THEN
18       blinkLeft := lamp_off ||
19       blinkRight := cycleMaxLampStatus(onCycle)
20     ELSE
21       blinkLeft := cycleMaxLampStatus(onCycle) ||
22       blinkRight := lamp_off
23   END
24 END; ...

```

Listing 1.1. Blink Management Machine

3.3 Integrating with Pitman Controller

The `PitmanController` reacts to changes in the sensors and sets the logical blinking state using the operations provided by the `BlinkLamps` machine. For example, Listing 1.2 shows how this is achieved using two parallel operation calls, one to `SET_EngineOn` from the `Sensors` machine and one to `SET_BlinkersOn` from `BlinkLamps` shown above in Listing 1.1. It is also interesting to note that the second operation call is wrapped into a conditional.

```

1  ENV_Turn_EngineOn =
2  BEGIN
3    SET_EngineOn ||
4    IF pitmanArmUpDown : PITMAN_DIRECTION_BLINKING &
5       hazardWarningSwitchOn = switch_off THEN
6       SET_BlinkersOn(pitman_direction(pitmanArmUpDown), continuousBlink)
7    END
8  END;
```

Listing 1.2. Pitman Controller Machine Event

3.4 Modelling Time

We have modelled time as a discrete integer variable representing elapsed time in milliseconds. The `GenericTimers` machine partially shown in Listing 1.3 manages a set of deadlines. The parameter `TIMERS` specifies a set of timers for which one can associate individual deadlines using the `AddDeadline` operation. Time can be advanced using the `IncreaseTime` operation, but it cannot proceed beyond a deadline, thus forcing events associated with the deadlines to be executed first. This is a typical scheme to model time in B, but variations and different approaches (e.g., using a controller event executed at fixed intervals).

For our modelling we needed two timers: one for the blinking phases and one for the management of “tip blinking”. Future extensions of the model will probably require additional deadlines which can be easily added.

```

1  MACHINE GenericTimers(TIMERS)
2  ...
3  INVARIANT
4    curTime : NATURAL &
5    curDeadlines : TIMERS +-> NATURAL
6  OPERATIONS
7    AddDeadline(timer, deadline) = PRE timer:TIMERS & deadline:NATURAL THEN
8      curDeadlines(timer) := curTime+deadline
9    END;
10   IncreaseTime(delta) = SELECT delta:NATURAL &
11     (curDeadlines/={} => curTime+delta <= min(ran(curDeadlines))) THEN
12     curTime := curTime + delta
13   END; ...
```

Listing 1.3. Generic Timer Machine for Simulation

The complete system model `PitmanController.TIME` then combines the untimed pitman controller with the `GenericTimers`. The Listing 1.4 shows how a deadline is set for the tip blinking event. When the deadline expires without the `pitmanArmUpDown` sensor changing the tip blinking is converted to a regular direction blinking.

```

1 MACHINE PitmanController_TIME
2 SETS    PTIMERS = {blink_deadline, tip_deadline}
3 INCLUDES Blinking,
4          GenericTimers(PTIMERS)
5 ...
6 OPERATIONS
7   ENV_Pitman_Tip_blinking_start(newPos) =
8   SELECT newPos : PITMAN_TIP_BLINKING &
9     newPos /= pitmanArmUpDown THEN
10     ENV_Pitman_Tip_blinking_short(newPos, pitman_direction(newPos)) ||
11     AddDeadline(tip_deadline, 500)
12   END;
13 ...

```

Listing 1.4. Pitman controller with time

The above version of `GenericTimers` in Listing 1.3 is good for simulation or replaying traces with explicit timing information such as the ones accompanying [7] (see Appendix A). As `curTime` is unbounded, however, it leads to an infinite state space for model checking. Hence we also developed a second “drop-in-replacement” of this machine, which has no `curTime` variable: the deadlines are always rescaled as if the current time was 0 (see Listing 1.5). In our case this was sufficient to produce a finite state model. Model checking the full system with ProB takes about 2.3s, generating 2095 states and 16472 transitions. (See [12] or [9] for related ways of model checking timed systems in state-based formal methods.)

```

1 MACHINE GenericTimersMC(TIMERS)
2 ...
3 INVARIANT
4   curDeadlines : TIMERS +-> NATURAL
5 OPERATIONS
6   AddDeadline(timer, deadline) = PRE timer:TIMERS & deadline:NATURAL THEN
7     curDeadlines(timer) := deadline
8   END;
9   IncreaseTime(delta) = SELECT delta:NATURAL &
10     (curDeadlines/={} => delta <= min(ran(curDeadlines))) THEN
11     curDeadlines := %x.(x:dom(curDeadlines)|curDeadlines(x)-delta)
12   END;
13 ...

```

Listing 1.5. Generic timer machine for model checking

4 Systems Modelling with Classical B and Translation to EventB

When translating a classical B composition such as the one in Fig. 2, one has to linearise the inclusion into a refinement chain. As you can see in Fig. 3, we have chosen the `BlinkLamps` as the top-level Event-B machine. In the translation, we had to split certain B operations into multiple events (as RODIN does not provide an IF-THEN-ELSE). E.g., the operation `SET_BlinkersOn` from Listing 1.1 is translated into two events: `SET_LeftBlinkersOn` and `SET_RightBlinkersOn`.

Let us now look at the `PitmanController` classical B machine, including `Sensors` and `BlinkLamps`. To translate this inclusion into refinement, we construct a *superposition* refinement of `BlinkLamps`, i.e., we add new variables and

Fig. 3. Hierarchy of final Rodin model, translated from classical B model in Fig. 2

do not remove existing variables.³ Moreover, as Event-B does not allow operation calls, we need to encode them using event refinement, more precisely refinement using the RODIN `extends` keyword which only adds actions and guards to an existing abstract event.

Let us look again at the operation `ENV_Turn_EngineOn` in Listing 1.2 from Sect. 3.3 above. It calls `SET_EngineOn` from `Sensors` and conditionally calls `SET_BlinkersOn` from `BlinkLamps`. This operation is translated into three events *refining* either `skip` or extending `SET_LeftBlinkersOn` or `SET_RightBlinkersOn`:

```

Event ENV_Turn_EngineOn_Noblink
  where
    grd1: engineOn = FALSE ∧ keyState = KeyInsertedOnPosition
    grd2: pitmanArmUpDown ∉ PITMAN_DIRECTION.BLINKING ∨
          hazardWarningSwitchOn = switch_on
  then
    act1: engineOn := TRUE
  end
Event ENV_Turn_EngineOn_BlinkLeft extends SET_LeftBlinkersOn
  where
    grd11: engineOn = FALSE ∧ keyState = KeyInsertedOnPosition
    grd12: pitmanArmUpDown ∈ PITMAN_DIRECTION.BLINKING
    grd13: pitman_direction(pitmanArmUpDown) = left_blink
    grd14: hazardWarningSwitchOn = switch_off
    grd15: rem = continuousBlink
  then
    act11: engineOn := TRUE
  end
Event ENV_Turn_EngineOn_BlinkRight extends SET_RightBlinkersOn
  where
    grd11: engineOn = FALSE ∧ keyState = KeyInsertedOnPosition
    grd12: pitmanArmUpDown ∈ PITMAN_DIRECTION.BLINKING
    grd13: pitman_direction(pitmanArmUpDown) = right_blink
    grd14: hazardWarningSwitchOn = switch_off
    grd15: rem = continuousBlink
  then
    act11: engineOn := TRUE
  end
  
```

One can see that the translation has resulted in code duplication, making it more tedious to adapt the model. A similar issue occurs at the next refinement

³ Machine inclusion in classical B can only add variables, not remove them.

level, where we introduce timing. As again we cannot refine multiple components, the code for the timer logic gets interspersed and duplicated in multiple events. Here we show just two instances in the `PitmanController.TIME_MC` model, which show how the guards and actions are replicated in multiple events.

```

Event  TIME_BlinkerOn extends TIME_BlinkerOn
  any   delta
  where
    grd21:  $\delta \in \mathbb{N}$ 
    grd22:  $\text{blink\_deadline} \in \text{dom}(\text{curDeadlines})$ 
    grd23:  $\delta = \text{curDeadlines}(\text{blink\_deadline})$ 
    grd24:  $\delta = \min(\text{ran}(\text{curDeadlines}))$ 
  then  actTm:  $\text{curDeadlines} := (\lambda x \cdot x \in \text{dom}(\text{curDeadlines}) \setminus \{\text{blink\_deadline}\} \mid$ 
            $\text{curDeadlines}(x) - \delta) \cup \{\text{blink\_deadline} \mapsto 500\}$ 
  end
Event  TIME_BlinkerOff extends TIME_BlinkerOff
  any   delta
  where
    grd21:  $\delta \in \mathbb{N}$ 
    grd22:  $\text{blink\_deadline} \in \text{dom}(\text{curDeadlines})$ 
    grd23:  $\delta = \text{curDeadlines}(\text{blink\_deadline})$ 
    grd24:  $\delta = \min(\text{ran}(\text{curDeadlines}))$ 
  then  actTm:  $\text{curDeadlines} := (\lambda x \cdot x \in \text{dom}(\text{curDeadlines}) \setminus \{\text{blink\_deadline}\} \mid$ 
            $\text{curDeadlines}(x) - \delta) \cup \{\text{blink\_deadline} \mapsto 500\}$ 
  end

```

If we want to modify the time management we need to edit multiple events in a consistent fashion, which is tedious and error-prone. The classical B model called the same operation in both cases; changing details about how time is handled just means changing or substituting the `GenericTimers` machine. Hence the conversion to Event-B in our methodology (Sect. 2) is delayed until the model has sufficiently stabilised.

Proof Statistics and Insights. We checked the correctness of our translation by checking the state spaces generated by PROB is identical for those three models that can be compared. The results are summarised in Table 1.

Table 1. Size of classical B and Event-B model state spaces and model checking times

Model	States	Transitions	Model checking time
BlinkLamps (B)	31	417	0.10 s
BlinkLamps (Event-B)	31	417	0.07 s
PitmanController (B)	74	514	0.13 s
PitmanController (Event-B)	74	514	0.10 s
PitmanController.TIME_MC (B)	2096	16742	2.39 s
PitmanController.TIME_MC (Event-B)	2096	16742	2.00 s

In summary, the translation resulted in a model with many more events and more duplication, but was worth it in the end. The proving process was relatively painless, the RODIN prover was easy to use. The proof statistics can be found in

Table 2, and the proving process itself helped to uncover a few interesting invariant properties and some issues in our sub-components (which were not reachable by the model checking, but would have appeared if the subcomponents were used differently). Also, the generation and discharging of the well-definedness proof obligations provides another safety guarantee, which is not exhaustively covered by model checking with PROB.

Table 2. Rodin proof statistics

Element Name	Total	Auto	Manual	Rev.	Und.
ABZ2020_v4	228	204	24	0	0
c0	2	1	1	0	0
c1	1	0	1	0	0
ctimers	0	0	0	0	0
BlinkLamps	72	64	8	0	0
PitmanController	107	100	7	0	0
PitmanController2_TIME	19	17	2	0	0
PitmanController2_TIME_MC	27	22	5	0	0

LTL Model Checking. As the RODIN model contains a series of events for one particular action, like turning the engine on or off, we wanted to check that it is always possible to turn the engine on or off and that there is exactly one event which describes the system’s evolution. For this we introduced the relative deadlock and controller state LTL properties in [8], which we used here. One particular property is the following one, which was violated by earlier versions of our translation:

```

1  G not(deadlock(ENV_Turn_EngineOn_BlinkLeft,
2                      ENV_Turn_EngineOn_BlinkRight, ENV_Turn_EngineOn_Noblink,
3                      ENV_TurnEngineOff_Blink, ENV_TurnEngineOff_Noblink))

```

Listing 1.6. LTL Formula

5 VisB Visualization of the Light System

The core idea of our new PROB plugin called VISB is to have a lightweight visualization engine which can be easily maintained and which make use of graphics generated with off-the-shelf editors. More concretely, VISB makes use of a SVG (scalable vector graphics) file and a JSON markup file. The markup files contains formal model expressions which specify attributes of objects which should be updated according to the current state of a formal model.

The present case study description [7] contains several appealing images, which we kindly obtained from the coordinators of the case study. Luckily, some were already in the SVG format and as such relatively little effort was needed to obtain a first visualisation (after about one hour the first visualisation was

Fig. 4. SVG graphics used for VISB without modifications

Fig. 5. Erroneous state uncovered via the visualisation

working). The main work was to combine the images into a single one, iron out a few quirks in the SVG files, and change the relevant object identifiers to mnemonic names and then write a JSON glue file referring to these identifiers. The original, unmodified SVG file is shown in Fig. 4 and the start of the JSON glue file in Listing 1.7.

As one can see, it specifies the original SVG file and provides instructions on how to update the attributes of objects (such as `stroke-opacity`) of particular objects (such as `A-right` corresponding to the front left light of the car). The value is given by a B expression which can reference the state of the formal model, in this case the `active_blinkers` variable. The glue file also specifies events which should be executed when certain objects are clicked upon.

```
1 {
2   "svg": "LichtUebersicht_v4.svg",
3   "items": [
4     {
5       "id": "A-right",
6       "attr": "stroke-opacity",
7       "value": "IF right_blink:active_blinkers THEN \"0.5\" ELSE \"1\" END"
8     },
9     ...
10  ]
11 }
```

Listing 1.7. Start of VisB JSON glue file

The effect of applying the VISB glue file to the original SVG can be seen in Fig. 5. This visualization shows the state of the actuators `blinkLeft` and `blinkRight`, as well as the state of the sensors `keyState`, `pitmanArmUpDown`, `hazardWarningSwitchOn` and `engineOn`. It also shows the internal state of the controller, e.g., the `active_blinkers` variable by changing the stroke-opacity and using a light orange fill. This state is an actual error that was spotted thanks to the visualisation: the hazard warning switch is on, but only the right blinkers are on. We can also see that the pitman arm is pushed up, but the controller correctly considers both left and right blinkers to be active, but somehow the blink management component has *not* set `blinkLeft` to the expected value (100).

VISB works for all of PROB's supported state-based formalisms (B, Event-B, Z, TLA+, Alloy), and as such we reused the same visualization for our classical B and Event-B models. A visualisation of the important states of the validation trace 7 from [7] can be found in Appendix A.

In conclusion, with relatively little effort it was possible to generate a visualisation from the graphics of the case study specification. In addition, the visualisation glue file was re-usable for a wide variety of models (several version of the classical B and Event-B models).

6 Description of Issues Uncovered

Some of the issues we found during implementing and especially during model checking the models are:

- Per specification, the system is not commutative. In ELS-13 it is stated that any tip-blinking is ignored or will be stopped by the hazard warning lights when hazard warning is active, respectively activated. So, we do have possible traces where at the same time hazard warning light is switched off and tip-blinking to either side is activated. Depending on which signal gets processed first, we either get three cycles of blinking or none at all. Same holds for switching on the engine and tip-blinking.
- In cruise control: while model checking we got shown errors in the activation of the cruise control as well as in the increasing and decreasing the desired speed, these errors seemed intended or at least directly caused by the specification. Depending on the version of the specification, there were different errors:
 - The desired speed in an activated cruise control could be lowered arbitrarily in previous versions (up to 1.14). This led to invariant violations while model checking, as by pushing down the lever from a desired speed of 0, it was further decreased.
 - The same invariant violation was caused by increasing the desired speed further than said maximum.
 - In the current version (1.17), those are fixed, as some upper and lower thresholds are specified, as well as a minimum velocity when activating it. But as of now, when no previous desired speed is set, the scs is required to be activated at current speed (scs-2) as long as the current speed is above or at 20 km/h. Here an activation at a speed of above 200 km/h should be possible according to the specification, thus directly violating the invariant of being in the range of 1..2000.

- As we have a reverse gear specified (but our speed being in the range of 0..5000), obviously the speed is measured in absolutes. That means that activation of the speed control system is also possible when in reverse gear (as long as going over 20 km/h in reverse). With possible high previous desired speed, this should either wreck our motor or gearbox fast if it keeps going and accelerating in reverse.
- There was a problem with the distance in adaptive cruise control, as the distance could be set to f.e. 3 s – with speeds of 240 km/h we would have had to measure distances that are outside our possible range. The latest specification solves this issue by setting a maximum target speed of 200 km/h.
- Ambient lights: els-19 states the low beam headlight should stay active for 30 more seconds with the interval being reset for example any time a door is opened or shut. We don't have sensors to measure that, as we only have a sensor to state if any door is open. So basically, this requirement has to be changed to either the 30s cycle being reset continuously until the status of all doors is closed or to the cycle being reset only for the first door opened and respectively the last door shut.

7 Related Work and Conclusion

The importance of animation and visualization seems now accepted; see e.g. the applications [13] and earlier ABZ case studies [5,6,8]. The idea of exploratory phases has also been promoted elsewhere, e.g., in [11].

The modelling approach described in Sect. 2 has been successful. Along the way we gained insights about the relationship between classical B machine inclusion and Event-B refinement. We were successful in re-using existing graphics to generate a custom graphical visualization, which helped us validate the model and allowed us to spot one error at least (cf. Fig. 5).

A weak point of our approach is the manual translation from classical B to Event-B in phase 3. An automated translation of a subset of classical B to Event-B would reduce human effort and avoid errors in the translation. Another alternative would be to stay with ATELIER-B and use its Event-B syntax, which does allow a more powerful substitution language. However, we were much more comfortable with the proving environment of RODIN, hence the effort to translate the models was worth it for us here (and for an industrial system modeling project undertaken by the first author). Finally, why did we not simply do all of the modelling in Rodin to start with? The reasons are the limited tool support for structuring, editing and sharing of models. Future improvements to RODIN could obviate the need to conduct the phases 1 and 2 of Sect. 2 in classical B. These improvements would require improved editing⁴ with git integration, improved model composition and decomposition features.

Acknowledgements. We thank Frank Houdek, Alexander Raschke and anonymous reviewers for their useful feedback.

⁴ CamilleX developed by the Univ. of Southampton is step in the right direction.

A Trace 7 from Case Study Specification with VisB

For convenience we include a condensed form of the visualisation of trace 7 from <https://github.com/hhu-stups/abz2020-models>. This shows trace 7 of the case study specification graphically as replayed with PROB and VisB.

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
3. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006). https://doi.org/10.1007/11901433_32
4. Comptier, M., Leuschel, M., Mejia, L.-F., Perez, J.M., Mutz, M.: Property-based modelling and validation of a CBTC zone controller in Event-B. In: Collart-Dutilleul, S., Lecomte, T., Romanovsky, A. (eds.) RSSRail 2019. LNCS, vol. 11495, pp. 202–212. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-18744-6_13
5. Hansen, D., et al.: Using a formal B model at runtime in a demonstration of the ETCS hybrid level 3 concept with Real Trains. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) ABZ 2018. LNCS, vol. 10817, pp. 292–306. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91271-4_20
6. Hoang, T.S., Snook, C., Ladenberger, L., Butler, M.: Validating the requirements and design of a hemodialysis machine using iUML-B, BMotion studio, and co-simulation. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 360–375. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_31

7. Houdek, F., Raschke, A.: Adaptive exterior light and speed control system (2019). <https://abz2020.uni-ulm.de/case-study>
8. Ladenberger, L., Hansen, D., Wiegard, H., Bendisposto, J., Leuschel, M.: Validation of the ABZ landing gear system using proB. *STTT* **19**(2), 187–203 (2017). <https://doi.org/10.1007/s10009-015-0395-9>
9. Lamport, L.: Real-time model checking is really simple. In: Borriane, D., Paul, W. (eds.) *CHARME 2005*. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005). https://doi.org/10.1007/11560548_14
10. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. *STTT* **10**(2), 185–203 (2008). <https://doi.org/10.1007/s10009-007-0063-9>
11. Oda, T., Araki, K., Larsen, P.G.: A formal modeling tool for exploratory modeling in software development. *IEICE Trans.* **100–D**(6), 1210–1217 (2017)
12. Rehm, J., Cansell, D.: Proved development of the real-time properties of the IEEE 1394 root contention protocol with the event b method. In: *ISoLA*, pp. 179–190 (2007)
13. Yang, F., Jacquot, J.-P., Souquières, J.: The case for using simulation to validate Event-B specifications. In: Leung, K.R.P.H., Muenchaisri, P. (eds.) *APSEC*, pp. 85–90. IEEE (2012)