Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Modeling of a Speed Control System using Event-B ⋆

Amel Mammar[1], **Marc Frappier**[2]

[1]SAMOVAR, Institut Polytechnique de Paris, Télécom SudParis, France
[2]GRIC, Département d'informatique, Faculté des sciences, Université de Sherbrooke, Québec, Canada

Received: date / Revised version: date

Abstract. This paper presents an Event-B model of a speed control system, a part of the case study provided in the ABZ2020 conference. The case study describes how the system regulates the current speed of a car according to a set of criteria like the driver's desired speed, the position of a possible preceding vehicle but also a given speed limit that the driver must not exceed. For that purpose, this controller reads different information from the available sensors (key state, desired speed) and takes adequate actions by acting on the actuators of the car's speed according to the information read. To formally model this system, we adopt a stepwise refinement approach with the Event-B method. We consider most of the features of the case study. All proof obligations of the invariant properties have been discharged using the Rodin provers. Our model has been validated using ProB by applying the different provided scenarios. This validation has permitted us to point out and correct some mistakes, ambiguities and oversights contained in the first versions of the case study.

Key words: Speed control system, Event-B method, Refinement, Verification

1 Introduction

The case study, proposed in the context of the ABZ2020 conference, is composed of two parts: *Adaptive Exterior Light* and *Speed Control Systems*. Since the whole case study is quite lengthy/complex and the two parts are only loosely coupled as stated in the description document, we chose to handle each part in a separate paper. The present paper deals with the speed control system whereas a companion paper considers the adaptive exterior light system [**?**].

The goal of the speed control system is to regulate the current speed of a car according to a set of criteria like the driver's desired speed, the position of a possible preceding vehicle but also a given speed limit that the driver must not exceed. The system can behave according to two options: the first one, called *normal* mode, regulates the speed independently from any preceding vehicle, the second option, called *adaptive* mode, takes into account the distance of the vehicle ahead by maintaining a safety distance. The driver has the possibility to choose which option to activate at any given moment. Like a controller, in both options, the system reads different information from the available sensors (key state, desired speed, the preceding vehicle position) and takes adequate actions by sending commands to the actuators of the car's speed according to the information read.

This paper describes the formal modeling of the speed control system using the Event-B method and its refinement technique. This technique permits to master the complexity of a system by gradually introducing its different elements/characteristics. After discussing the different requirements to model and the modeling strategy, the Event-B models have been developed by the first author. Her experience in the formal specification and verification of railway interlocking systems, in collaboration with Thales and RATP, helped her in this task. She has also developed the Event-B models for the previous ABZ case studies [**?**,**?**]. The development of the Event-B models took about one month and half (full-time) and has been done under the Rodin platform [**?**] that provides editors, provers and several other plugins for various tasks like animation and model checking

Send offprint requests to: Amel Mammar, E-mail: amel.mammar@telecom-SudParis.eu

⋆ This work was supported in part by the DISCONT ANR French project and NSERC (Natural Sciences and Engineering Research Council of Canada).

with ProB [?]. Our Event-B model considers most of the requirements except the following ones:

- Requirements SCS-21, SCS-27 and SCS-28 (emergency braking) which are not precise enough to be formalized;
- Requirement SCS-30 is not a functional requirement, it is related to the visual interface;
- Requirement SCS-43 is related to the light control system, which is specified in another model [?].

We use ProB in order to animate the built models with two purposes: (1) exhibiting the problematic sequences of events, called scenarios, that violate the invariant along with the expected value of each variable; (2) validating the specification by simulating the desired scenarios, provided in the requirement document, in order to be sure that we have specified the right system. These animation/validation phases permitted to exhibit several ambiguous/informal descriptions and also some errors in the specification (see Section 5.1). To fix them and to remove such ambiguities, we have intensively exchanged with the authors of the requirement document [?], Frank Houdek in particular.

Our approach to model the control speed system follows the four-variable model of Parnas and Madey [?] that distinguishes two groups of variables *environment* and *controller* variables (see Section 2.1). The first group denotes the elements that are outside the controller; they are the elements whose states are read by the sensors and to which commands are sent through the actuators. The second group represents the input and the outputs of the system. In that case, the approach to model a control system in Event-B consists in modeling at each refinement step an environment or a controller variable in order to master the complexity of the system.

1.1 The Event-B method

Event-B [?] is the successor of the B method [?] permitting to model discrete reactive systems using mathematical notations. The complexity of a system is mastered thanks to the refinement concept that allows to gradually introduce the different parts that constitute the system, starting from an abstract model and gradually refining it into a more concrete one.

An Event-B specification is made of two types of elements: *contexts* and *machines*. A context describes the static part of an Event-B specification; it consists of sets (user-defined types) S and constants C together with axioms A that specify their properties:

CONTEXT *Cont*
Sets S
Constants C
Axioms A
END

The dynamic part of an Event-B specification is included in a machine that defines variables V and a set of events E. The possible values that the variables are allowed to take are specified using an invariant, denoted Inv, written using a first-order logic formula on the state variables:

MACHINE *Name*
SEES *Cont*
Variables V
Invariants Inv
Events E

Each event has the following form:

ANY X
WHERE G
THEN Act
END

An event can be triggered if it is enabled, i.e. the condition G, named guard, holds. When more than one event is enabled, only one, chosen nondeterministically, is triggered. When an event is triggered, its actions Act are applied over variables. In this paper, we restrict ourselves to the *becomes equal* action, denoted by: $x := e$.

The triggering of each event must maintain the invariant. To this aim, proof obligations are generated. For each event, we have to establish that [?]:

$$\forall\, S, C, V, X.\ (A \wedge G \wedge Inv \wedge Act \Rightarrow Inv')$$

where Act is the before-after predicate of the actions of an event and Inv' is the invariant applied to the after values.

Refinement is a process of enriching or modifying a model in order to augment the functionality being modeled, and/or explain how some purposes are achieved. Both Event-B elements *context* and *machine* can be refined. A context can be extended by defining new sets S_r and/or constants C_r together with new axioms A_r. A machine is refined by adding new variables and/or replacing existing variables by new ones V_r that are typed with an additional invariant Inv_r called *gluing invariant* that permits to link both the variables V and V_r. New events can also be introduced to implicitly refine a **skip** event, which means that a new event cannot modify existing variables, but only new variables. In this paper, the refined events have the same form:

ANY X_r
WHERE G_r
THEN Act_r
END

To prove that a refinement is correct, we have to establish the following two proof obligations [**?**]:

- *guard refinement*: the guard of the refined event must be stronger than the guard of the abstract one:

$$\forall(S, C, S_r, C_r, V, V_r, X, X_r).\\ (A \wedge A_r \wedge Inv \wedge Inv_r \Rightarrow (G_r \Rightarrow G))$$

- *simulation*: the effect of the refining action must simulate the effect of the abstract one. In other words, the effect of the refining action must be included in that of the abstract one:

$$\forall\ (S,\ C,\ S_r,\ C_r,\ V,\ V_r,\ X,\ X_r).\\ (A \wedge A_r \wedge G_r \wedge Inv \wedge Inv_r \wedge Act_r \Rightarrow\\ Act \wedge Inv'_r)$$

To discharge the different proof obligations, Rodin [**?**] offers an automatic prover but also the possibility to plug in additional external provers like the SMT [**?**] and AtelierB provers [**?**] that we use in this work. Both provers offer automatic and interactive modes to discharge the proof obligations.

1.2 The ProB model checker

ProB [**?**] is, an animator and an automatic model checker, originally developed for the verification and validation of software development based on the B language [**?**]. Developed at the University of Düsseldorf, Germany, starting from 2003, ProB implements an automatic model-checking technique to check LTL (Linear Temporal Logic) [**?**] and CTL (Computational Tree Logic) [**?**] properties against a B specification. The core of ProB is written in Prolog. The purpose of ProB is to be a comprehensive tool in the area of formal verification methods. Its main functionalities can be summarized as follow:

1. ProB can find a sequence of events that, starting from a valid initial state of the machine, moves the machine into a state that violates its invariant.
2. Given a valid state, ProB can exhibit the event that causes the invariant to be violated.
3. ProB supports the animation of B/Event-B specifications to permit the user to simulate different scenarios from a given starting state that satisfies the invariant. Through a graphical user interfaces implemented in Tcl/Tk and Java UI [**?**], the animator provides the user with: (1) the current state, (2) the history of the event triggering that has led to the current state and (3) a list of all the enabled events, along with proper argument instantiations. In this way, the user does not have to guess the right values for the event arguments.

1.3 Contributions

The contributions of this paper with respect to [**?**] are as follows:

- a more detailed description of the modeling strategy used (see Section 2);
- re-expression of some properties (like SCS-41) as invariants that can be formally proved (see Section 3.2.1);
- additional examples and explanations about the verification phase using model-checking and scenarios (see Section 5);
- a comparison with similar solutions using different approaches/formal languages (see Section 6).

1.4 The structure of the paper

The rest of this paper is structured as follows. Section 2 describes our modeling strategy. Section 3 introduces our model in more details. Section 4 describes the validation and verification of our model. Section 5 identifies the defects found in the requirements document provided for the case study [**?**], and the adequacy of the Event-B method for constructing a model of this case study. Section 6 compares our solution with other solutions of this case study. We conclude in Section 7.

2 Requirements and modeling strategy

This section describes the principles of control systems and the general Event-B architecture of their modeling.

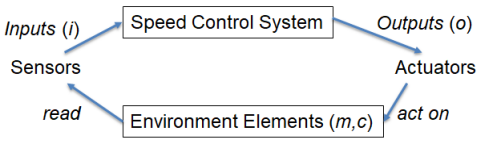

Fig. 1. The speed control system behaviour

2.1 Control abstraction

The speed control system subject of this paper can be seen as a control system that interacts with its environment through a set of sensors, which provide it with information about the state of the physical elements, and a set of actuators that are used to transmit the adequate orders to these elements (see Figure 1). In this paper, we use the concepts of controlled/monitored variables to describe the environment elements [**?**]. A sensor measures the value of some environment elements m, called a *monitored* variable (*e.g.*, the state of the ignition key), and provides this measure (*e.g.*, whether the key is inserted or not) to the software controller as an *input* variable i. The software controller can influence the environment by sending commands, called *output* variable o to actuators. An actuator influences the value of some characteristics of the environment, called a *controlled* variable c. Variables m and c are called *environment variables*. Variables i and o are called *controller variables*. Finally, a controller has its own internal state variables to perform computations. In this case study, we use Event-B state variables to represent both environment and controller variables. We do not model sensor or actuator failures.

A well-known architecture of a control system is a control loop that reads all input variables at once, at a given moment, and then computes all output variables in the same iteration. But, it can be also viewed as a continuous system that can be interrupted by any change in the environment represented by a new value sent by a sensor. In this paper, we see the controller as a distributed system; each sub-system (environment element) is associated to a given sensor. In that case, the system reacts to each single modification of the sensor and to the progression of time:

WHILE *true* **DO**
  IF *timer of an action is expired* **THEN**
    execute timing action
  END
  Read Inputs from some sensors
  IF *the value of a sensor changes* **THEN**
    Process the sensor modification
    Send Outputs to the actuators
  END
END

Our modeling approach is more abstract, as it is common in the Event-B style of system modeling. We define one event for each input variable change, which allows for a more modular specification that is easier to prove. This is closer to an interrupt-driven control system. Our Event-B abstraction is also a reasonable abstraction for a control loop, considering that in most cases, a single input variable changes between two control loop iterations. The control loop can be derived from our specification by merging all events and defining dynamic priorities between events in order to avoid any starvation problem.

2.2 Modeling structure

The Event-B project modeling the case study is composed of four levels depicted by Figure 2:

- Machine M0 models the current speed of the studied car independently from any preceding vehicle and also without giving any condition on its evolution. It sees a context C0 that defines the allowed values for this speed;
- Machine M1 introduces the physical elements that are manipulated by the driver and that have an impact on the current speed of the car. These elements include gas/brake pedal, key, cruise control lever. The description of these elements is defined in the context C1 that extends the context C0. Machine M1 describes how the position of each of these elements evolves depending on its current position;
- Machine M2 models the desired speed whose allowed values are defined in the context C2 that extends the context C1. In that machine, we model the activation of the normal/adaptive cruise control and also the traffic sign detection that has an impact on the value of the desired speed according to the requirements (SCS-36, SCS-39). It is worth noting that some events, like those related to the traffic sign detection, are introduced in M1 even if this aspect is really dealt with in the machine M2. Indeed, these events need to modify some variables that are introduced in M1 and, as noted before, a new event cannot modify a variable defined in a previous refinement level;
- Machine M3 specifies the different aspects that depend on or impact the desired/current speed, like the safety distance from the preceding vehicle, which depends on the speed of that vehicle but also the faults that can happen on the radar system. Information on the radar together with the safety distance are defined in the context C3 that extends the context C2.

3 Model details

This section describes the main modeling elements that characterize our specification. The emphasis is put on

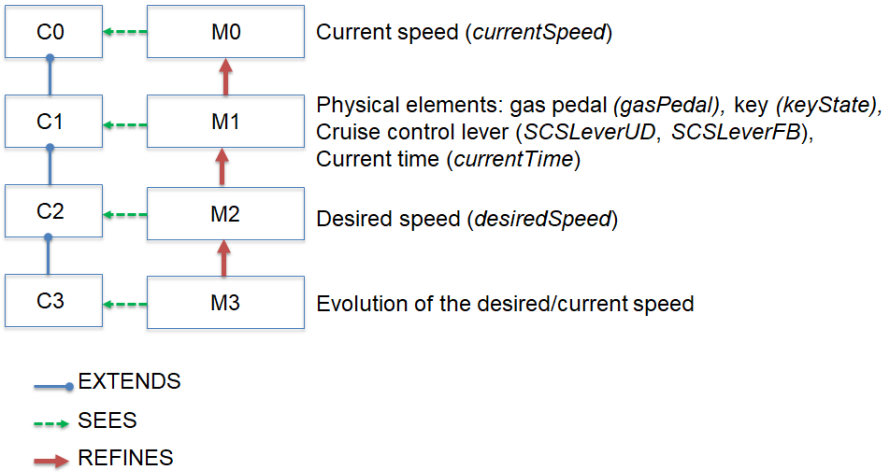

Fig. 2. Structure of the Event-B model

the parts that needed effort and those illustrating the key properties of the system: evolution of the speed and maintaining a safety distance. The complete archive of the Event-B project is available in [?]. Table 3 relates the components of our model with the requirements listed in [?]. As one can remark, some requirements are modeled as invariants whereas others are dealt with in the adequate events.

The following subsections give more details about the modeling of some elements of the case study.

3.1 Refinement level 0 (M0+C0)

Machine M0 models the current speed of the observed car independently from any preceding vehicle and also without giving any condition on its evolution. To this aim, it sees a context C0 that defines the constant *rangeSpeed* representing the range of values for the speed[1]:

$$rangeSpeed = 0..5000$$

The current speed is defined by the following invariant in the machine M0:

$$currentSpeed \in rangeSpeed$$

Machine M0 defines a single event *updateVehicleSpeed* to set the current speed of the car as follows:

Event updateVehicleSpeed $\widehat{=}$
  any
    val
  where
    grd1: $val \in rangeSpeed$
  then
    act1: $currentSpeed := val$
  end

[1] Let us note that the upper bound stands for 500 km/h and the resolution is 1/10 km/h

3.2 Refinement level 1 (M1+C1): physical elements

This machine refines the machine M0 by introducing the different elements that impact the current speed of the car. This includes the physical elements that the driver manipulates, the radar system that gives the distance to the nearest obstacle, the key that permits to start the engine but also the time progression since it makes some variables evolve, like the desired speed. For that purpose, several variables/invariants are introduced to model how the position of the physical elements evolves depending on their current positions. In this level, the machine M1 defines the following variables:

- *currentTime*: it represents the current time of the system. This variable is useful for modeling timed aspects.
- *keyState* (resp. *keyStateP*): it denotes the current (resp. previous) position of the key.
- *SCSLeverUD* (resp. *SCSLeverUDP*): it models the current (resp. previous) position (*Upward*/ *Downward*) of the cruise lever.
- *SCSLeverFB* (resp. *SCSLeverFBP*): it models the current (resp. previous) position (*Forward*/ *Backward*) of the cruise lever.
- *brakePedal*: it denotes the deflection of the brake pedal.
- *gasPedal*: it denotes the deflection of the gas pedal from the neutral position.
- *speedLimiterSwitchOn*: it denotes the position of a button that when it is pushed indicates that the cruise control lever is used as as speed limiting lever.
- *rangeRadarState*: it denotes the state of the radar sensor which can be *Ready*, *Dirty* or *NotReady*. As stated in the case study description, the behavior of the radar is the same in both states *Dirty* and *NotReady*, thus we model this state by a Boolean variable: **TRUE** for *Ready* and **FALSE** for both states *Dirty* and *NotReady*.

Requirements [?]	Component	Invariant/Event
SCS-1, SCS-31, SCS-38	M2	*inv3*
SCS-2, SCS-31, SCS-38	M2	*inv4*
SCS-3, SCS-12, SCS-13, SCS-16, SCS-17, SCS-31, SCS-38	M2	*inv5* and *inv6*
SCS-4, SCS-19, SCS-31, SCS-38	M2	*inv7*
SCS-5, SCS-19, SCS-31, SCS-38	M2	*inv8*
SCS-6, SCS-19, SCS-31, SCS-38	M2	*inv9* and *inv10*
SCS-7, SCS-19, SCS-31, SCS-38	M2	*inv11*
SCS-8, SCS-19, SCS-31, SCS-38	M2	*inv12*
SCS-9, SCS-31, SCS-38	M2	*inv13*
SCS-10, SCS-31, SCS-38	M2	*inv14*
SCS-11, SCS-31, SCS-38	M2	*inv15*
SCS-14	M3	*inv17* and *inv18*
SCS-15	M3	*inv19*
SCS-18	M3	*inv8*
SCS-21		not covered
SCS-22	M3	*inv7*
SCS-23	M3	*inv9*, *inv10* and *inv11*
SCS-24	M3	*inv12*
SCS-25	M3	*inv14*
SCS-26	M3	*inv15*
SCS-27-SCS-28		not covered
SCS-29	M1	*inv9*
SCS-30		not covered since it is related to the graphical user interface
SCS-32, SCS-33, SCS-34	M2	*inv20*
SCS-35	M1	*inv8*
SCS-36, SCS-37, SCS-39	M2	*inv22*, *inv26*, *inv25*
SCS-40 and SCS-41	M2	Event moveKey and progress
SCS-42	M3	*inv19*
SCS-43		not covered since the light system is not included

Table 1. Cross-reference between the components of our model and the requirements of [**?**]

– *nextTest*: it memorises the time of the next radar testing. As stated in the requirements document, the self-test of the radar is performed every 10 minutes.
– *lastTest*: it memorises the time when the previous test has been performed. This variable is reset to 0 when the key is not in the ignition.

In the following, we give details about the key, the radar system, the time progression and also the cruise control lever. We model these elements in the same refinement level since the behaviour of the radar depends on time progression and the position of the key.

3.2.1 Modeling the key and the radar

To model the key, we have defined in the context C1, a set *keyStates* to describe all the states of the key[2]:

partition(keyStates, {*NoKeyInserted*}, {*KeyInserted*}, {*KeyInIgnitionOnPosition*})

We also define a constant *KeyMoves* to denote the authorized transitions for a key:

$$\begin{aligned} KeyMoves = \{\ & NoKeyInserted \mapsto KeyInserted, \\ & KeyInserted \mapsto KeyInIgnitionOnPosition, \\ & KeyInIgnitionOnPosition \mapsto KeyInserted, \\ & KeyInserted \mapsto NoKeyInserted\ \} \end{aligned}$$

In the machine M1, the variable *keyState* represents the current state of the key, the variable *keyStateP* contains the previous state of the key and the authorized transitions are specified in the invariant inv2:

$$\begin{gathered} keyStateP \mapsto keyState \in KeyMoves \\ \vee \\ keyStateP = keyState \end{gathered}$$

Similarly, the state of the radar system is modeled by a Boolean variable *rangeRadarState*. This variable is initialized to **FALSE** since the ignition is off at the beginning; then its state is updated every 10 minutes. In

[2] $\textbf{partition}(A,\ A_1,\ \ldots,\ A_n) \Leftrightarrow \begin{pmatrix} A = A_1 \cup \ldots \cup A_n \\ \wedge \\ \forall A_i, A_j . (A_i \neq A_j \Rightarrow A_i \cap A_j = \varnothing) \end{pmatrix}$

comparison to the conference paper, we have improved the modeling of this requirement by modeling it as an invariant that can be formally proved. For that purpose, we defined the following invariants in the machine M1:

– the variable *lastTest* is equal to the current time when the system (re)starts and remains always less or equal to the current time:

$$\begin{array}{l} lastTest \leq currentTime \wedge \\ keyStateP \neq KeyInIgnitionOnPosition \wedge \\ keyState = KeyInIgnitionOnPosition \\ \Rightarrow \\ lastTest = currentTime \end{array}$$

– the variable *lastTest* is reset to 0 when the system is off:

$$\begin{array}{l} keyState \neq KeyInIgnitionOnPosition \\ \Rightarrow \\ lastTest = 0 \end{array}$$

– Variables *nextTest* and *lastTest* are either both equal to 0 (at the initialisation of the system) or separated by *updateDur* units of time ($u.t$). where the constant *updateDur* is defined in the context C1 as being equal to *600* (10×60) since the update is performed each ten minutes. When the system (re)starts, the variable *nextTest* is set to *currentTime* + *updateDur* to permit testing the system. This variable is also updated to *currentTime* + *updateDur* each time such deadline is reached.

$$\begin{array}{c} (nextTest = 0 \wedge lastTest = 0) \\ \vee \\ (nextTest \text{ - } lastTest = updateDur) \end{array}$$

To guarantee that these invariants are preserved, adequate actions should be added to the events that update the variable *keyState*, that is, the event moveKey that models the behavior of the key. Hereafter, we give an excerpt of the Event-B modeling of this event:

Event moveKey $\widehat{=}$
any
valkey
radstate
where
grd1: $keyState \mapsto valkey \in KeyMoves$
grd2: $radstate \in$ **BOOL**
grd3: ...
then
act1: $keyState := valkey$
act2: $keyStateP := keyState$
act3: $rangeRadarState := radstate$
act4: $nextTest :=$ **if** $(valkey = KeyInIgnitionOnPosition)$ **then** $currentTime + updateDur$ **else** $nextTest$
act5: $lastTest :=$ **if** $(valkey = KeyInIgnitionOnPosition)$ **then** $currentTime$ **else** 0
act6: ...
end

For readability, the actions act4 and act5 in the event moveKey use a conditional **if** c **then** $x := v1$ **else** $x := v2$ construct which is not provided as a native Event-B notation. To overcome this limitation, the models developed under Rodin use the following idiom to a conditional **if** c **then** $x := v1$ **else** $x := v2$ construct:

$$x := \{\textbf{TRUE} \mapsto v1, \textbf{FALSE} \mapsto v2\}(\textbf{bool}(c))$$

where the term $\{\textbf{TRUE} \mapsto v1, \textbf{FALSE} \mapsto v2\}$ denotes a function, so it is evaluated at point $\textbf{bool}(c)$. Operator $\textbf{bool}(c)$ evaluates formula c and returns a result of the predefined set $BOOL = \{\textbf{TRUE}, \textbf{FALSE}\}$.

Let us note that another solution to model the behavior of the key would be to define a separate event for each key state transition. However, we did not choose this option since only the transitions from/to the state KeyInIgnitionOnPosition affect the behavior of the system. Moreover, grouping these two transitions in a single event allows for the factorisation of the proof steps that would be common to the two events.

3.2.2 Modeling time progression

In cyber-physical systems, like the one we deal with, the controller reads the values of the sensors and sends commands to actuators on a periodical basis called a *cycle*, whose duration affects the safety correctness of the system. In general, the value of time progression is the shortest time between two consecutive sensors' updates. For simplicity, we assume in this paper that the shortest time between two consecutive sensors' updates is greater than one second.

As the speed evolves with time progression, we specify the event progress that models this progression as a refinement of the event updateVehicleSpeed:

Event progress $\widehat{=}$
refines updateVehicleSpeed
any
val
radstate
where
grd1: $val \in rangeSpeed \wedge radstate \in$ **BOOL**
grd2: $keyState \neq KeyInIgnitionOnPosition \vee nextTest \neq currentTime + 1 \implies radstate = rangeRadarState$
...
then
act1: $currentSpeed := val$
act2: $currentTime := currentTime + 1$
act3: $keyStateP := keyState$
act4: $rangeRadarState := radstate$
act5: $nextTest :=$ **if** $(keyState = KeyInIgnitionOnPosition \wedge nextTest = currentTime + 1)$ **then** $currentTime + 1 + updateDur$ **else** $nextTest$
act6: $lastTest :=$ **if** $(keyState = KeyInIgnitionOnPosition \wedge nextTest = currentTime + 1)$ **then** $currentTime + 1$ **else** $lastTest$

end

Guard *grd2* specifies that the value of the state of the radar system does not change if the there is no time progression or the key is not in the state *KeyInIgnitionOnPosition*, otherwise it is nondeterministically chosen (Guard *grd1*: $radstate \in \mathbf{BOOL}$).

As this event is guarded, we have to establish that the conjunction of these guards is satisfiable in order to ensure that the progression of the time is never blocked. To this aim, we have introduced a theorem TH on this event to ensure that there exists at least a combination of values for the parameters that makes all the guards are satisfied:

$$\exists\ val,\ radState, \ldots,\ .\ (grd_1 \wedge \ldots \wedge grd_n)$$

From the practical point of the view, the proof of this theorem makes the Rodin's prover crash with a Java heap space error since the number of cases to consider is huge. Each case is related to a particular value of some variables (*keyState*, *SCSLeverUDP*, etc.). To overcome this problem, we have broken this big theorem into several and simpler ones according to these values. These sub-theorems are of the following form:

$$\begin{array}{l} P_i \Rightarrow \mathrm{TH} \\ \bigvee_i P_i \end{array}$$

For instance to prove the theorem TH, we have introduced three theorems Th1, Th2 and Th3 depending on the position of the key:

$$\begin{array}{l} \mathrm{Th1} = keyState = KeyInIgnitionOnPosition \Rightarrow \\ \qquad \exists\ val,\ radState, \ldots,\ .\ (grd_1 \wedge \ldots \wedge grd_n) \\ \mathrm{Th2} = keyState \neq KeyInIgnitionOnPosition \Rightarrow \\ \qquad \exists\ val,\ radState, \ldots,\ .\ (grd_1 \wedge \ldots \wedge grd_n) \\ \mathrm{Th3} = keyState = KeyInIgnitionOnPosition \vee \\ \qquad\qquad keyState \neq KeyInIgnitionOnPosition \end{array}$$

Moreover, we model all the timed aspects (like speed and the radar stated which is tested every 10 minutes) in this event (see Section 3.3) to make them evolve together with the time, while other aspects (like key) are modeled in separate events.

3.2.3 Modeling the cruise control lever

The cruise control lever is modeled by the variable *SCSLeverUD* (*Upward*/*Downward*) and the following typing invariant:

$$SCSLeverUD \in SCSLeverPositions$$

where *SCSLeverPositions* is a given set defined in Context C1 seen by M1:

$$\begin{array}{l} \mathbf{partition}(SCSLeverPositions,\ Upward,\ Downward, \\ \qquad \{Backward\},\ \{Forward\},\ \{Neutral\}) \\ \mathbf{partition}(Upward,\ \{Upward5\},\ \{Upward7\}) \\ \mathbf{partition}(Downward,\ \{Downward5\},\ \{Downward7\}) \end{array}$$

For each of these elements, invariants are defined in the machine $M1$ to specify the authorized position changes together with the event that models them. The following invariant *invPos* states that the cruise control lever cannot directly move from an *Upward* position to a *Downward* position bypassing the *Neutral* position[3]:

$$\begin{array}{l} SCSLeverUD \in Upward \Rightarrow \\ \qquad SCSLeverUDP \in Upward \cup \{Neutral\}) \\ \qquad\qquad \wedge \\ (SCSLeverUD \in Downward \Rightarrow \\ \qquad SCSLeverUDP \in Downward \cup \{Neutral\}) \end{array}$$

As one can remark, the above invariant uses an extra variable *SCSLeverUDP* to model the previous position of the cruise control lever. In the next section, we show that this kind of variables is also relevant for modeling some requirements that need to make reference to the current and previous states of the system.

Machine $M1$ defines the event moveSCSLeverUD that models the cruise control lever movements where *grd2* permits to make the invariant *invPos* preserved after the execution of this event. Basically, the guard *grd2* is obtained from *invPos* by replacing *SCSLeverUDP* and *SCSLeverUD* by *SCSLeverUD* and *valSCS* respectively. The first term of the disjunction, $SCSLeverUD = SCSLeverUDP$, is not considered since the event aims at changing the position of the lever as stated, in the guard *grd*, with $valSCS \neq SCSLeverUD$. The term $SCSLeverUD = SCSLeverUDP$ is relevant in the expression of the invariant since it should be fulfilled by all the events that do not modify the position of the lever.

Event move$SCSLeverUD$ $\widehat{=}$
  any
    $valSCS$
  where
    grd1: $valSCS \in Upward \cup Downward \cup \{Neutral\}$
      $\wedge\ valSCS \neq SCSLeverUD$
    grd2: $SCSLeverUD \neq Neutral \implies$
      $(SCSLeverUD \in Upward \wedge valSCS \in Upward)$
      $\vee$
    $(SCSLeverUD \in Downward \wedge valSCS \in Downward)$
      $\vee$
      $(valSCS = Neutral)$
  then
    act1: $SCSLeverUD := valSCS$
    act2: $SCSLeverUDP := SCSLeverUD$
    ...: ...
  end

Let us note that a similar variable *SCSLeverFB* is defined for modeling *Forward*/*Backward* movement of the cruise control level along with its associated events.

[3] Another solution would be to define a constant *SCSLeverUDPos* to denote the possible transitions of the cruise control lever as we did for the key.

3.3 Refinement level 2 (M2+C2): desired speed

This machine describes how the desired speed evolves according to the requirements (SCS-1 to SCS-12) by moving the cruise control lever into different positions. We also model the activation of the normal/adaptive cruise control as described in the document. To this aim, these additional variables have been introduced:

- *desiredSpeed* (resp. *desiredSpeedP*): it denotes the desired speed (resp. the previous one) whose value cannot exceed 2000 (200 km/h). The null value (0) is used to model the absence of a desired value;
- *lastTimeSCSLeverUD*: it denotes the time of the last update of the variable *SCSLeverUD*;
- *adapContr*/*normContr*: it denotes the adaptive/normal mode for the cruise control; *adapContrP*/ *normContrP* represent the previous values.

In the rest of this section, we describe the activation of the cruise control and how the desired speed is calculated by distinguishing different cases.

3.3.1 Activation of the cruise control

The activation of the cruise control mode as specified in the requirement document [**?**] is modeled by the following invariant *CruiseControlKind*, where value 1 (resp. value 2) denotes the normal (resp. adaptive) cruise control mode:

Invariant *CruiseControlKind*

$$
\begin{array}{l}
\quad normContr = \textbf{TRUE} \\
\Leftrightarrow \\
\quad (\\
\qquad (SCSLeverFB = Forward \wedge \\
\qquad\ SCSLeverFBP \neq Forward \wedge \\
\qquad\ (currentSpeed \geq speedActiv \vee desiredSpeed \neq 0)) \\
\quad \vee \\
\qquad (normContrP = \textbf{TRUE} \wedge \\
\qquad\ SCSLeverFB \neq Backward) \\
\quad) \wedge \\
\quad cruiseControlMode = 1 \wedge \\
\quad brakePedal = 0
\end{array}
$$

This invariant states that when the normal mode is selected for the cruise control and the brake pedal is not activated, the normal cruise control is activated in two cases:

1. (i) the cruise control lever moves to the *Forward* position while the current speed is greater than *speedActiv* or the desired speed is not null. Constant *speedActiv* is defined in the context C2 to be equal to 200 (20 km/h),
2. (ii) it is already activated and the cruise control lever is not put in the *Backward* position.

3.3.2 Modeling the desired speed

To model the desired speed whose evolution depends on the time, we store the last time, *lastTimeSCSLeverUD*, when the cruise control lever has been in the up/down positions. Thus, requirements SCS-4 and SCS-7 are modeled as follows. Requirement SCS-4 specifies that, while the cruise control is activated, the desired speed increases by 10 (1 km/h) the first time the cruise control lever is put in position *Upward5* whereas Requirement SCS-7 states that the desired speed continues to increase by 10 (1 km/h) by each second as long as the cruise control lever stays in that position for more than 2 seconds. Variable *lastdesiredSpeed* represents the desired speed when the lever has been moved into a given position.

$$
\begin{array}{l}
\quad SCSLeverUDP \neq Upward5 \wedge \\
\quad SCSLeverUD = Upward5 \wedge \\
\quad (adapContrP = \textbf{TRUE} \vee normContrP = \textbf{TRUE}) \\
\Rightarrow \\
\quad desiredSpeed = \textbf{min}(\{desiredSpeedMax, \\
\qquad\qquad\qquad\qquad desiredSpeedP + 10\})
\end{array}
$$

and

$$
\begin{array}{l}
\quad (normContr = \textbf{TRUE} \vee adapContr = \textbf{TRUE}) \wedge \\
\quad SCSLeverUD = Upward5 \wedge \\
\quad currentTime \text{ - } lastTimeSCSLeverUD \geq 2 \\
\Rightarrow \\
\quad desiredSpeed = \textbf{min}(\{desiredSpeedMax, \\
\qquad desiredSpeedP + \\
\qquad (currentTime \text{ - } lastTimeSCSLeverUD \text{ - } 1) \times 10\})
\end{array}
$$

Let us give more explanation about the last invariant. Expression *currentTime* - *lastTimeSCSLeverUD* - 1 permits to update the desired speed immediately after 2 seconds, this is why we subtract 1 second not 2 seconds. To make these invariants preserved, we have refined the moveSCSLeverUD event according to the requirement SCS-4 but also the progress event with respect to the requirement SCS-7. Event progress for instance is refined by adding the following guards that calculate the new desired speed *despeed* by distinguishing different cases according to the position of the control lever and the time elapsed since its last position change: $currentTime + 1$ - $lastTimeSCSLeverUD \geq 2$. This value *despeed* is then assigned to the variable *desiredSpeed* by the action $desiredSpeed := despeed$. In the following different cases, the term $currentTime + 1$ denotes the after-value of *currentTime* when the event progress is observed:

1. **Case 1.** If the cruise control is not activated, there is no desired speed.

   $normContr = \textbf{FALSE} \wedge adapContr = \textbf{FALSE}$
   $\Rightarrow$
   $despeed = 0$

2. **Case 2.** If the cruise control is active with the lever either in the neutral position or the time elapsed from the last movement of the lever is less than 2 seconds then, the desired speed does not change.

$(normContr = \textbf{TRUE} \lor adapContr = \textbf{TRUE}) \land$
$(SCSLeverUD = Neutral \lor$
$currentTime + 1 \text{ - } lastTimeSCSLeverUD < 2)$
$\Rightarrow$
$despeed = desiredSpeed$

3. **Case 3.** If the cruise control is active with the lever in the *Upward5* position for more than 2 seconds, the desired speed increases by 10 ($1\ km/h$) every second: SCS-7.

$(normContr = \textbf{TRUE} \lor adapContr = \textbf{TRUE}) \land$
$SCSLeverUD = Upward5 \land$
$currentTime\text{- } lastTimeSCSLeverUD \geq 2$
$\Rightarrow$
$despeed = \textbf{min}(\{desiredSpeedMax, lastdesiredSpeed +$
$(currentTime\text{- } lastTimeSCSLeverUD) \times 10\}$

4. **Case 4.** If the cruise control is active with the lever in the *Upward7* position for more than 2 seconds, the desired speed increases to the next ten's place after each 2 seconds: SCS-8.

$(normContr = \textbf{TRUE} \lor adapContr = \textbf{TRUE}) \land$
$SCSLeverUD = Upward7 \land$
$currentTime + 1 \text{ - } lastTimeSCSLeverUD \geq 2$
$\Rightarrow$
$despeed = \textbf{min}(\{desiredSpeedMax,$
$lastdesiredSpeed \div 100 \times 100 +$
$((currentTime + 1 \text{ - } lastTimeSCSLeverUD) \div 2) \times 100\})$

5. **Case 5.** If the cruise control is active with the lever in the *Downward5* position for more than 2 seconds, the desired speed decreases by 10 ($1\ km/h$) every second: SCS-9.

$(normContr = \textbf{TRUE} \lor adapContr = \textbf{TRUE}) \land$
$SCSLeverUD = Downward5 \land$
$currentTime +1\text{- } lastTimeSCSLeverUD \geq 2$
$\Rightarrow$
$despeed = \textbf{max}(\{10, lastdesiredSpeed \text{ -}$
$(currentTime \text{ - } lastTimeSCSLeverUD) \times 10\})$

6. **Case 6.** If the cruise control is active with the lever in the *Downward7* position for more than 2 seconds, the desired speed decreases to the next ten's place after each 2 seconds: SCS-10.

$(normContr = \textbf{TRUE} \lor adapContr = \textbf{TRUE}) \land$
$SCSLeverUD = Downward7 \land$
$currentTime + 1 \text{ - } lastTimeSCSLeverUD \geq 2$
$\Rightarrow$
$despeed = \textbf{max}(\{10, (lastdesiredSpeed \div 100) \times 100 \text{ -}$
$((currentTime + 1 \text{ - } lastTimeSCSLeverUD) \div 2) \times 100\})$

3.4 Refinement level 3 (M3+C3): other elements

In this machine, we model the different aspects that depend on or impact the desired/current speed, like speed-dependent safety distance and the speed of the preceding vehicle. Moreover, we model the faults that can happen on the radar system as described in the requirement document [**?**]. For this aim, the machine M3 introduces, among others, the following variables:

- *accVeh*: it denotes the acceleration/deceleration of the car.
- *safetyDistance*: it denotes the distance in seconds that is allowed with respect to the car ahead;
- *securedistanceToHead*: it denotes the distance in meters between the considered car and the car ahead;
- *rangeRadarSensor*: it denotes the distance in meters to some obstacle detected in the travel corridor;
- *speedOfHead* (resp. *speedOfHeadP*): it denotes the (resp. previous) speed of the car ahead.

In this machine, we model the evolution of the speed according to its acceleration/deceleration *accVeh*. Let us note that since the Event-B language does not support real numbers, we model the current speed as an integer amount that evolves according to the usual equation $V = \gamma \times t + V_p$, where γ represents the acceleration/deceleration of the vehicle, $t = 1$ the time progression and V_p the previous speed. Another alternative to overcome the lack of reals in the Event-B language is to define or reuse an existing theory plugin that models them [**?**]. However, this plugin does not permit to evaluate expressions involving real operands. For instance, the expression (2×3) cannot be reduced to 6. Moreover, floating point numbers are not considered. So, at this level, the parameter *val* of the event progress is made equal to:

- if the car is decelerating:

$accVeh \leq 0 \Rightarrow$
$val = \textbf{max}(\{0,$
$(accVeh + currentSpeed \times 10 \div 36) \times 36 \div 10\})$

- if the car is accelerating without speed limitation:

case 1: *disabled cruise control*

$accVeh \geq 0 \land$
$speedLimiterSwitchOn = \textbf{FALSE} \land$
$((normContr = \textbf{FALSE} \land adapContr = \textbf{FALSE})$
$\lor desiredSpeed = 0)$
$\Rightarrow$
$val = (accVeh +$
$currentSpeed \times 10 \div 36) \times 36 \div 10$

case 2: *enabled cruise control*

$accVeh \geq 0 \land$
$speedLimiterSwitchOn = \textbf{FALSE} \land$
$(normContr = \textbf{TRUE} \lor adapContr = \textbf{TRUE}) \land$
$desiredSpeed \neq 0$
$\Rightarrow$
$val = \textbf{min}(\{desiredSpeed,$
$(accVeh + currentSpeed \times 10 \div 36) \times 36 \div 10\})$

A similar expression is specified for the case with speed limitation.

Moreover, the machine M3 introduces two new events turnHead and VehicHeadDetect to model respectively the selection of a safety level by turning the cruise control lever head and the detection of a preceding vehicle by getting its speed that is relevant for determining the speed-dependent safety distance and also to make the system decelerate if it is necessary. Event VehicHeadDetect for instance is specified as follows:

Event VehicHeadDetect $\widehat{=}$
  any
    val stv secdis speh accv
  where
    grd1: $val \in rangeRadarSensorValues$
    grd2: $rangeRadarState = \mathbf{FALSE} \Leftrightarrow val = 255$
    grd3: $speh \in rangeSpeed$
    grd4: $speh \leq speedActiv \wedge$
      $speedOfHead > speh \wedge speh \neq 0 \wedge$
      $adapContr = \mathbf{TRUE} \wedge val \notin \{0{,}255\}$
      $\Longrightarrow$
      $secdis = 25 \times (currentSpeed \times 10 \div 36)$
    grd5: $speh = 0 \wedge currentSpeed = 0 \wedge$
      $adapContr = \mathbf{TRUE} \wedge$
      $val \notin \{0{,}255\}$
      $\Longrightarrow$
      $secdis = 2$
    ...: ...
  then
    act1: $rangeRadarSensor := val$
    act2: $speedOfHead := speh$
    act3: $securedistanceToHead := secdis$
    act4: ...
  end

where:

- *val* represents the distance between the observed car and a possible preceding vehicle as provided by the radar;
- *stv* denotes the value of the speed received as an input by the system;
- *brk* represents the future brake pressure;
- Parameter *secdis* is the safety distance to the car ahead;
- *speh* denotes the speed of the car ahead.

Guard grd2 states that such a value should be equal to 255 *iff* the radar system is not ready. Guards grd4 and grd5 permit to calculate the new value for the speed-dependent safety distance according to the following part of the requirement SCS-23: *If the speed of the preceding vehicle is 20 km/h or below, the distance is set to (2.5s × currentSpeed), down to a standstill. When both vehicles are standing the absolute distance is regulated to 2m.*

Already existing events of M2 are refined in M3 in a similar way by calculating the value of the different variables. For instance, the desired speed should be updated when a traffic sign is detected, the speed-dependent safety distance is updated when the current speed is modified or the speed of a preceding vehicle changes. More details can be found in [**?**].

4 Validation and verification

To ensure the correctness and validate the Event-B models, we have proceeded in three steps detailed hereafter. It is worth noting that these steps are performed in an iterative manner, that is: we first check the specification for invariant violations using (non-exhaustive) model-checking, then we verify that the system behaves as expected and last we discharge the generated proof obligations. Of course, any modification to the specification requires repeating these verification steps.

4.1 Model checking of the specification

We used ProB as a model checker to ensure that the invariants of each machine are preserved after the execution of each event, that is, there is no sequence of events that violates an invariant. Basically, when an invariant violation is found, ProB emits a sequence of events that, starting from a valid initial state of the machine, leads to a state that violates the related invariant. Such specification errors can be due to a guard/action missing, to an incorrect specification of the invariant, but sometimes also to an incorrect requirement, that is the intended system behavior cannot satisfy the desired property (i.e., the requirements are inconsistent).

In this particular case study, some properties require the use of auxiliary variables to store the previous value of a variable (*SCSLeverUDP* and *keyStateP* for instance). In an initial version of the event moveSCSLeverUD, the action act2 had been omitted, causing the violation of the invariant invPos for the trace execution depicted by Table 2. Indeed, the last line of Table 2 states that the cruise control lever moves from the position *Upward5* to the position *Downward5* without passing by the position *Neutral*.

Let us note that even if no invariant violation is found by the tool, there may still exist scenarios violating the invariant that the tool cannot find due to their complexity (scenarios with several steps) and/or the timeout on the model checking process. This is why a proof phase should be performed to ensure that the specification is invariant-preserving.

4.2 Validation with scenarios

This step aims at verifying that we have built the right model whose behaviors conform to the desired ones as described by the scenarios of the specification document. For that purpose, the animation capability of ProB is

Step	Event	SCSLeverUDP	SCSLeverUD
1	Initialisation	*Upward5*	*Upward5*
2	SCSLeverUD	*Upward5*	*Neutral*
3	SCSLeverUD	*Upward5*	*Downward5*

Table 2. Execution trace violating an invariant

used to simulate the different scenarios provided in the case study. Each scenario consists of a set of actions performed by the user along with the expected value of each variable after executing an action. Thus, to simulate a scenario, we have mapped each action to an event of our model in order to check the values of the variables against to the desired ones. This step allows us to point out some flaws/ambiguities in the initial release of the requirements document. For instance, the initial examples provided to illustrate the requirements SCS-5—SCS-9 were incorrect with respect to the requirements. Initially, the requirement SCS-7 was as follows:

> *Current speed is* 57 *km/h* $\rightarrow$ *after holding* 2 *seconds, desired speed is set to* 58 *km/h, after holding* 3 *seconds, desired speed is set to* 59 *km/h, after holding* 4 *seconds, desired speed is set to* 60 *km/h, ...*

It has been rephrased as follows since the desired speed increase by 1 *km/h* as soon as the cruise control lever is moved to level 2 up to the first resistance level (5°) and the (adaptive) cruise control is activated:

> *Current* ***desired*** *speed is* 57 *km/h* $\rightarrow$ ***new desired speed is*** 58 ***km/h (due to Req. SCS-4)****, after holding* 2 *seconds, desired speed is set to* 59 *km/h, after holding another second, desired speed is set to* 60 *km/h, after holding another second, desired speed is set to* 61 *km/h, ...*

In addition, in some places like SCS-7—SCS-9, the term "target speed" is used instead of "desired speed". Moreover during the reviewing process of this paper, one reviewer drew our attention that SCS-10 wrongly uses the term *increases* instead of *decreases*. All these aspects have been discussed with the case study authors because we are not specialists of the domain. Let us note that we have faced some difficulties to simulate the provided scenarios since no information is provided on how the controller calculates the acceleration at each step. So, we have done our best to "simulate" these values without any guidance about their suitability, reliability. Moreover, we think that more scenarios are required to help us validate our models and also for a better understanding of the system, considering that the informal description is imprecise on some points and that the system includes several elements whose behaviors are interrelated and impact each other.

4.3 Proof of the specification

This last phase aims at ensuring the correctness of the specification by discharging all the proof obligations generated by Rodin to prove that the invariants are preserved by each event, but also that the guard of each refined event is stronger than that of the abstract one. Figure 3 provides the proof statistics[4] of the case study: 547 proof obligations have been generated, of which 38% (211) were automatically proved by the various provers. The remaining proof obligations were discharged interactively, since they needed the use of external provers, like the Mono Lemma prover, that has shown to be very useful for arithmetic formulas, even if we had to add some theorems on min/max operators. For instance, a min/max of a finite set is an element of the set, and also on the transitivity property of the comparison operator ($\geq$, $\leq$, ...). For this particular case study, the interactive proofs were not difficult but required several steps especially those using the min/max operators and Boolean expressions (**bool**(..)) that need to distinguish several cases; this represents about 50 proof obligations.

5 Discussion

This section reports on some points regarding the choices made during the Event-B modeling of the speed control system.

5.1 Feedback on the requirements document

In the validation step of the formal modeling of the requirements document [**?**] (see Section 4.2), using the different scenarios provided in the requirement document, we have discovered a number of ambiguities and some contradictions that led us to question ourselves about the meaning of some requirements. Not being specialist of the domain, we have communicated these to the authors of the requirements document, and a number of revisions were produced, following our comments. Our discussion and exchange led to the modification/revision of a set of requirements to make them clearer and consistent. A detailed list of these elements are described in the last version (*i.e.*, 1.17) of the requirements document:

[4] The reader may get different statistics when opening our GitHub archive of the project. There is currently a problem in Rodin when it reports proof statistics. Some manual POs are reported as automatic when a project archive is imported. The Rodin team has been notified of this problem.

Proof Control | Statistics × | Rodin Problems

Element Name	Tot.	Aut.	Man.	Rev.	Und.
SCS_1	**547**	**211**	**336**	**0**	**0**
C0	0	0	0	0	0
C1	0	0	0	0	0
C2	18	16	2	0	0
C3	0	0	0	0	0
M0	2	1	1	0	0
M1	116	100	16	0	0
M2	234	0	234	0	0
M3	177	94	83	0	0

Fig. 3. Rodin proof statistics of the case study

1. Correction of the examples in SCS-7, SCS-8 and SCS-9 since the values do not respect the requirements;
2. Modification of signal description *setVehicleSpeed* to make its meaning clearer;
3. Replacing "target speed" by "desired speed" in requirements SCS-7 and SCS-8;
4. Adjustment of the maximum acceleration and deceleration values in SCS-20, SCS-22;
5. Stating that SCS-23 applies when the speed is less or equal to 20 km/h;
6. Clarification of priority between adaptive cruise control and emergency braking assistant in case of brake activation in SCS-28;
7. The signal *SCSLever* has been split into signals *SCSLeverForthBack* and *SCSLeverUpDown* with their corresponding positions (states) and the possible transitions between them.

As already well-known, the use of a formal method does not only permit to built a correct system, but it also helps to clarify the requirements document by detecting omissions, ambiguities and errors.

It is worth noting that the requirement document describes the behavior of a car in terms of its speed and its safety distance to the car ahead to avoid their collision. It would be interesting to be able to verify that these cars never collide. For that purpose, we should model the position evolution of each car and prove that their positions do not overlap. A similar proof has been done by the authors [**?**] for train collisions in the ERTMS case study submitted for ABZ2018 [**?**].

5.2 Modeling temporal properties

As stated before, a number of requirements relate a change in some variable V to the value of some other variable X. We model this by introducing VP, that denotes the previous value of V, and adding an invariant $I_{V,X}$ of the form $V \neq VP \Rightarrow \Phi(X)$. This entails that an event that modifies V must also update VP, but events that only modify X must also update VP using $VP := V$, in order to prove $I_{V,X}$. This is quite cumbersome and tedious to manage. We have also explored the modeling of such properties using LTL formulas, however, Event-B does not support the expression of LTL formula as part of the specification. Moreover, the verification of these formulas using ProB does not terminate for our model because of the size of the state space. We think that it would be interesting to investigate existing tools/approaches that could help us specify this kind of properties in a simpler manner. An example of such tools is the Event-B state machines plugin [**?**,**?**] that produces Event-B events from a state machine including their guards that specify the requirements modeled by the state machine but without producing the related invariants. For our case study, this plugin would be used to ease the modeling of temporal properties and automatically generate the corresponding Event-B events. However, this plugin makes it slightly more difficult to understand the meaning of the generated guards. It would be also interesting to extend the Event-B language to include the notion of a previous value of a variable to be used in invariants.

5.3 Dealing with variable requirements

The specification of the case study contains the following variability aspects that impact the behavior of the system:

- *cruiseControlMode*: specifies the operation mode of the cruise control, that is, *normal* or *adaptive*.
- *trafficSignDetectionOn*: specifies if the traffic sign detection is enabled or not.

These variability aspects make some behaviors possible for specific values and not enabled for others. For in-

stance, if the mode *adaptive* is selected by the user in the instrument cluster settings menu, he/she can let the adaptive cruise control adapt the desired speed without having to use the gas/brake pedal.

To deal with these variability aspects in EVENT-B, we defined two constants: $cruiseControlMode \in \{1,2\}$ and $trafficSignDetectionOn \in \mathbf{BOOL}$ in the context C2 (see Section 3.3) as specified in the requirement document [**?**]. Then, we have expressed the invariants corresponding to the related requirement by including conditions on the values of these constants. For instance, the requirement SCS-3 that describes the activation of the adaptive cruise control is modeled by:

$$
\begin{array}{l}
\quad adapContr = \mathbf{TRUE} \\
\iff \\
\quad (\\
\quad\ (SCSLeverFB = Forward \land \\
\quad\quad SCSLeverFBP \neq Forward \land \\
\quad\quad (currentSpeed \geq speedActiv \lor desiredSpeed \neq 0) \\
\quad\ ) \\
\quad \lor \\
\quad\ (adapContrP = \mathbf{TRUE} \land \\
\quad\ SCSLeverFB \neq Backward) \\
\quad) \land \\
\quad cruiseControlMode = 2 \land brakePedal = 0
\end{array}
$$

The above invariant is complementary to the invariant *CruiseControlKind* specified in Section 3.3.1 that describes the activation of the normal cruise control.

6 Comparison

Few papers have dealt so far with the speed control system case study, potentially because it is preceded in [**?**] by another case study on an exterior light system (ELS) that is quite challenging in itself, so most people may have addressed ELS first.

MISRA C, a set of guidelines that describes a subset of the C language, is used in [**?**] to formally model the speed control case study. The different requirements and the dynamics of the system is encoded within these guidelines for the purpose of verification. The requirements are classified into two classes: *simple* and *complex*. The simple requirements are verified at first since they involve very small numbers of system elements and thus make their verification easier. Complex requirements are verified in a second step, using the CBMC model checker [**?**], since they aim at verifying the interaction between several components of the system. The authors report on some flaws/ambiguities but did not state how they dealt with them. Moreover, even if this approach has the advantage to directly produce the executable code, its correctness cannot be guaranteed since model checking on a limited scope does not ensure the absence of bugs.

ASM [**?**] are used in [**?**] to model this system following a refinement-based approach, very similar to our. A first abstract model is designed at the top of the development to present the overall objective of the system. This model is then refined to include more details about different elements. The scenarios provided in the specification document are used in order to validate by animation the developed models. The NUSMV [**?**] tool is also used to verify, by model-checking, some properties expressed as CTL/LTL formulas. However, the paper does not report on the errors/ambiguities detected in the specification.

7 Conclusion

This paper presents a formal modeling of a speed control system using the EVENT-B method. We have modeled most of the requirements, which permitted us to point out some ambiguities that we have discussed and clarified with the case study authors by rephrasing them. These ambiguities have been discovered during different development phases: formalization, proof and validation using the provided scenarios. This experience has confirmed that the formal modeling of a system greatly helps the designers clarify their requirements and detect errors early in development phases, which makes their correction cheaper. However, reading formal models in general can be difficult and requires having some mathematical background. We think that this point can be improved by providing designers with a visual representation of the models using for instance the VISB [**?**] component of PROB. Such a visual representation would help designers have a better understanding of their formal models by visualizing the behavior of the system.

The main difficulty when modeling the speed control system is to determine the order in which elements should be introduced during the refinement especially since many elements are interdependent. Due to time constraints, we were unfortunately not able to explore the different decomposition plugins of RODIN that might produce smaller specification parts that would be easier to understand and maintain [**?**]. We plan to explore some decomposition techniques as future work. We strongly believe that EVENT-B should include modularization operators as native structuring mechanisms, like those of the B method. It would permit to have a modular specification from the early development phases, and it would make EVENT-B more suitable for the development of big and complex systems. Another point concerns the PROB plugin under RODIN that unfortunately does not permit to store an already simulated scenario, so one has to manually re-simulate each scenario; this is very time-consuming for long traces.

The work presented in this paper can also be extended by considering the remaining requirements that need more clarifications. Requirement SCS-21 for in-

stance needs more information on how the system can deduce that a deceleration of 3 m/s^2 is insufficient to prevent a collision without having any information about the acceleration of the preceding vehicle. Also, we think that more information should be provided on the internal variables like *setVehicleSpeed* that represents the automatic acceleration of the system in order to be able to build a more complete system. Moreover, through the different case studies proposed in the ABZ conference [?,?], we are now convinced of the need to improve the Event-B language to make it support the real numbers as a basic type. Its prover should be also extended to include more rules on arithmetic and set theories. Finally, it would be interesting to consider code generation from the last refinement level in order to obtain the implementation of the controller. For that purpose, we would need to implement all the nondeterministic substitutions, i.e. the **ANY** substitutions that chose values for input parameters. Moreover, parallel substitutions will be implemented by sequential ones by introducing auxiliary variables if needed.

Acknowledgments The authors would like to thank the case study authors, and Frank Houdek in particular, for his responsiveness and useful feedback during the modeling process when questions were raised or when ambiguities were found. The authors would also like to thank Michael Leuschel for his quick feedback on using ProB for this large case study. Since the first version of the paper, the very valuable reviewers' comments have helped us improve the quality of the paper. We would like to sincerely thank them.

References

1. Abrial, J.R.: The B-book - Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. Software Tools for Technology Transfer 12(6), 447–466 (2010)
4. Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: Modelling an Automotive Software-Intensive System with Adaptive Features Using ASMETA. In: Rigorous State-Based Methods - 7th International Conference ABZ. Lecture Notes in Computer Science, vol. 12071, pp. 302–317. Springer (2020)
5. Bendisposto, J., Geleßus, D., Jansing, Y., Leuschel, M., Pütz, A., Vu, F., Werth, M.: ProB 2-UI: A Java-Based User Interface for ProB. In: Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems). Lecture Notes in Computer Science, vol. 12863, pp. 193–201. Springer (2021)
6. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer (2003)
7. Butler, M., Maamria, I.: Practical Theory Extension in Event-B, pp. 67–81. Springer Berlin Heidelberg (2013)
8. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In: 25 Years of Model Checking - History, Achievements, Perspectives. Lecture Notes in Computer Science, vol. 5000, pp. 196–215. Springer (2008)
9. Clarke, E.M., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference TACAS. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)
10. Clearsy: `https://www.atelierb.eu/en/presentation-of-the-b-method/formal-proof-presentation/`
11. Event-B Consortium: SMT Solvers Plug-in. `https://wiki.event-b.org/index.php/SMT_Solvers_Plug-in`
12. Fotso, S.J.T., Frappier, M., Laleau, R., Mammar, A.: Modeling the Hybrid ERTMS/ETCS Level 3 Standard using a Formal Requirements Engineering Approach. International Journal on Software Tools for Technology Transfer 22(3), 349–363 (2020)
13. Hoang, T.S., Butler, M., Reichl, K.: The Hybrid ERTMS/ETCS Level 3 Case Study. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 251–261. Lecture Notes in Computer Science, Springer (2018)
14. Hoang, T.S., Dghaym, D., Snook, C., Butler, M.: A Composition Mechanism for Refinement-Based Methods. In: 22nd International Conference on Engineering of Complex Computer Systems ICECCS. pp. 100–109. IEEE Computer Society (2017)
15. Houdek, F., Raschke, A.: Adaptive Exterior Light and Speed Control System. `https://abz2020.uni-ulm.de/case-study#Specification-Document` (November 2019)
16. Krings, S., Körner, P., Dunkelau, J., Rutenkolk, C.: A Verified Low-Level Implementation of the Adaptive Exterior Light and Speed Control System. In: Rigorous State-Based Methods - 7th International Conference ABZ. Lecture Notes in Computer Science, vol. 12071, pp. 382–397. Springer (2020)
17. Leuschel, M., Butler, M.J.: ProB: An Automated Analysis Toolset for the B Method. International Journal on Software Tools for Technology Transfer 10(2), 185–203 (2008)
18. Leuschel, M., Bendisposto, J., Dobrikov, I., Krings, S., Plagge, D.: From Animation to Data Validation: The ProB Constraint Solver 10 Years On. In: Formal Methods Applied to Complex Systems: Implementation of the B Method, chap. 14, pp. 427–446. Wiley ISTE (2014)
19. Mammar, A., Frappier, M.: An Event-B Model of a Speed Control System. Available at `https://github.com/AmelMammar/SpeedControlSystem` (March 2023)
20. Mammar, A., Frappier, M., Fotso, S.J.T., Laleau, R.: A Formal Refinement-Based Analysis of the Hybrid ERTMS/ETCS Level 3 Standard. International Journal on Software Tools for Technology Transfer 22(3), 333–347 (2020)
21. Mammar, A., Frappier, M., Laleau, R.: An Event-B Model of an Automotive Adaptive Exterior Light System. In: Rigorous State-Based Methods - 7th Interna-

tional Conference ABZ. Lecture Notes in Computer Science, vol. 12071, pp. 351–366. Springer (2020)
22. Mammar, A., Laleau, R.: Modeling a Landing Gear System in Event-B. International Journal on Software Tools for Technology Transfer 19(2), 167–186 (2017)
23. Mammar, A., Frappier, M.: Modeling of a Speed Control System Using Event-B. In: Rigorous State-Based Methods - 7th International Conference ABZ. Lecture Notes in Computer Science, vol. 12071, pp. 367–381. Springer (2020)
24. Mammar, A., Frappier, M., Fotso, S.J.T., Laleau, R.: An Event-B Model of the Hybrid ERTMS/ETCS Level 3 Standard. In: Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference ABZ. Lecture Notes in Computer Science, vol. 10817, pp. 353–366. Springer (2018)
25. Mammar, A., Laleau, R.: Modeling a Landing Gear System in Event-B. In: ABZ 2014: The Landing Gear Case Study - Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z. Communications in Computer and Information Science, vol. 433, pp. 80–94. Springer (2014)
26. Event-B Consortium: `http://www.event-b.org/`
27. Nakahori, K., Yamaguchi, S.: A support Tool to Design IoT Services with NuSMV. In: IEEE International Conference on Consumer Electronics, (ICCE). pp. 80–83. IEEE (2017)
28. Parnas, D.L., Madey, J.: Functional Documents for Computer Systems. Science of Computer Programming 25(1), 41–61 (1995)
29. Pnueli, A.: The Temporal Logic of Programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence. pp. 46–57. IEEE Computer Society (1977)
30. ProB: `https://prob.hhu.de/`
31. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Transactions on Software Engineering and Methodology 15(1), 92–122 (2006)
32. Snook, C.: `http://wiki.event-b.org/index.php/Event-B_Statemachines`
33. Werth, M., Leuschel, M.: VisB: A Lightweight Tool to Visualize Formal Models with SVG Graphics. In: Raschke, A., Méry, D., Houdek, F. (eds.) Proceedings ABZ 2020. pp. 260–265. Lecture Notes in Computer Science, Springer (2020)

A An excerpt of the requirements

Requirement label	description
SCS-1	After engine start, there is no previous desired speed. The valid values for desired speed are from 1 km/h to 200 km/h.
SCS-2	When pulling the cruise control lever to ①, the desired speed is either the current vehicle speed (if there is no previous desired speed) or the previous desired speed (if already set).
SCS-3	If the current vehicle speed is below 20 km/h and there is no previous desired speed, then pulling the cruise control lever to 1 does not activate the (adaptive) cruise control.
SCS-4	If the driver pushes the cruise control lever to ② up to the first resistance level (5^0) and the (adaptive) cruise control is activated, the desired speed is increased by 1 km/h.
SCS-5	If the driver pushes the cruise control lever to ② above the first resistance level (7^0, beyond the pressure point) and the (adaptive) cruise control is activated, the desired speed is increased to the next ten's place.
SCS-6	Pushing the cruise control lever to ③ reduces the desired speed ac- cordingly to Req. SCS-4 and Req. SCS-5. The lowest desired speed that can be set by pushing the cruise control lever beyond the pressure point is *10 km/h*.
SCS-7	If the driver pushes the cruise control lever to ② with activated cruise control within the first resistance level (5^0, not beyond the pressure point) and holds it there for 2 seconds, the desired speed of the cruise control is increased every second by 1 km/h until the lever is in neutral position again.
SCS-8	If the driver pushes the cruise control lever to ② with activated cruise control through the first resistance level (7^0, beyond the pressure point) and holds it there for 2 seconds, the speed set point of the cruise control is increased every 2 seconds to the next ten's place until the lever is in neutral position again.
SCS-9	If the driver pushes the cruise control lever to ③ with activated cruise control within the first resistance level (5^0, not beyond the pressure point) and holds it there for 2 seconds, the desired speed of the cruise control is reduced every second by 1 km/h until the lever is in neutral position again.
SCS-10	If the driver pushes the cruise control lever to ③ with activated cruise control through the first resistance level (7^0, beyond the pressure point) and holds it there for 2 seconds, the speed set point of the cruise control is increased every 2 seconds to the next ten's place until the lever is in neutral position again.
SCS-11	If the (adaptive) cruise control is deactivated and the cruise control lever is moved up or down (either to the first or above the first resistance level, the current vehicle speed is used as desired speed.
SCS-12	Pressing the cruise control lever to ④ deactivates the (adaptive) cruise control. *setVehicleSpeed* = 0 indicates to the car that there is no speed to maintain.
SCS-20	If the distance to the vehicle ahead falls below the specified speed-dependent safety distance (see Req. SCS-24), the vehicle brakes automatically. The maximum deceleration is 3 m/s^2.
SCS-21	If the maximum deceleration of 3 m/s^2 is insufficient to prevent a collision with the vehicle ahead, the vehicle warns the driver by two acoustical signals (0.1 seconds long with 0.2 seconds pause between) and by this demands to intervene.
SCS-22	If the distance to the preceding vehicle increases again above the speed-dependent safety distance, the vehicle accelerates with a maximum of 1 m/s^2 until the set speed is reached.
SCS-23	If the speed of the preceding vehicle is 20 km/h or below, the distance is set to 2.5s × *currentSpeed*, down to a standstill. When both vehicles are standing the absolute distance is regulated to 2m. When the preceding vehicle is accelerating again, the distance is set to 3s × *currentSpeed*. This distance is valid until the vehicle speed exceeds 20 km/h, independent of the user's input via the distance level (turning the cruise control lever head).

SCS-24	By turning the cruise control lever head, the distance to be maintained to the vehicle ahead can be selected. Three levels are available: 2 seconds, 2.5 seconds and 3 seconds. The desired level only applies within the velocity window > 20 km/h. Below this level, the system autonomously sets the distance according to Req. SCS-23.
SCS-27	The emergency brake assistant must be available in the following speed windows: 0-60 km/h, for emergency braking to stationary obstacles, 0-120 km/h on moving obstacles.
SCS-28	The time necessary to perform braking to standstill is determined by the value for the maximum deceleration. If an object is ahead of the vehicle and the time until an impact is less or equal to the time until a standstill plus 3 seconds, three acoustic signals are given (0.1 seconds long with 0.05 seconds pause between) is issued and the brakes are activated by 20% (i.e. 1.2 m/s^2). If the time until an impact is less or equal to the time until a standstill plus 1.5 seconds, the brake is activated by 60% (i.e. 3.6 m/s^2. If the time until an impact is less or equal to the time until standstill then the brake is activated at 100% (i.e. 6 m/s^2). In case that both adaptive cruise control (see Req. SCS-20) and the emergency brake assistand request braking, the higher deceleration value shall be applied.
SCS-30	An active speed limit function of the cruise lever is indicated by an orange LED integrated in the control lever (realized in hardware).
SCS-36	Traffic sign detection is active, while adaptive cruise control is active and the driver has activated traffic sign detection in the instrument cluster.
SCS-39	If traffic sign detection recognizes Unlimited, the new desired speed is set to (i) 120 km/h, if the previous desired speed has been lower than 120 km/h. (ii) the desired speed d_{man}, where d_{man} is the last manually set desired speed that has been higher than 120 km/h.
SCS-41	If the radar sensor self-test device reports a fault (Dirty or NotReady), all systems depending on the distance to the vehicle must be suspended and the driver must be warned by an appropriate light in the instrument cluster (not part of this specification). In this case, the self-test of the radar system is restarted every 10 min.
SCS-43	If the system performs a brake action, the brake lights must be activated as if the brake pedal has been pressed by the driver (see light system specification).

Table 3: An excerpt of the requirements from [**?**]