International Journal on Software Tools for Technology Transfer (2020) 22:333–347
https://doi.org/10.1007/s10009-019-00543-1

A formal refinement-based analysis of the hybrid ERTMS/ETCS level 3 standard

Amel Mammar[1] · Marc Frappier[2] · Steve Jeffrey Tueno Fotso[2,3] · Régine Laleau[3]

Published online: 26 October 2019

Abstract
This paper presents a formal model of the case study proposed for the ABZ2018 conference, which concerns the Hybrid ERTMS/ETCS Level 3 Standard. This standard allows trains to communicate with a train supervisor to report their integrity and positions, thanks to an onboard train integrity monitoring system. The supervisor assigns trains a movement authority to control traffic and to avoid collisions. The standard also provides for trains that cannot communicate with the supervisor; these trains are detected by sensors on tracks and obey traffic signals set by the supervisor along the trackside. Using communication allows for a finer grain control of the tracks. Our model is derived using stepwise refinement with the EVENT- B method. We take into account the main features of the case study (VSS management, timers, ERTMS and non-ERTMS trains). Our model is decomposed into four refinements. All proof obligations have been discharged using the Rodin provers, except those related to the computation of the VSS state machine, which was found to be ambiguous (nondeterministic). Our model has been validated using PROB. The main safety property, which is that ERTMS trains do not collide, is proved. Our model focuses on the discrete control logic aspects of the case study.

Keywords Hybrid ERTMS/ETCS level 3 standard · Event-B method · Formal modeling and verification · Proof

1 Introduction

This paper proposes an EVENT- B [1] model of the hybrid ERTMS/ETCS level 3 case study [2–4] proposed for the ABZ2018 conference. The European Train Control System (ETCS) is the signalling and control component of the European Rail Traffic Management System (ERTMS). The approach presented in this paper uses the refinement technique of EVENT- B to master the complexity of the system. The different elements of the system are thus incrementally introduced in order to focus on a specific aspect. We have used the Rodin platform[1] [5], an Eclipse-based IDE for EVENT- B that provides effective support for refinement and mathematical proof. We have also used PROB[2] [6], a model checker/animator, to validate the specification by applying some scenarios, especially those described in [4]. The use of PROB proved very helpful, especially to detect some inconsistencies either in the specification document or in our understanding of it. After fixing the main errors, proof obligations generated by the Rodin Platform according to the EVENT- B method are discharged to prove properties of the EVENT- B specification. These properties include invariant preservation, deadlock freedom and event feasibility. Various invariants are proved, the most important being the absence of collision. Our model focuses on the discrete control logic aspects of the case study.

The development team is composed of four members (the authors of this paper). All authors have a good expertise in the EVENT- B method. Moreover, the first author has been involved in the formal specification and verification of railway interlocking systems with the collaboration of Thales and RATP. After a first discussion of the main

✉ Amel Mammar
amel.mammar@telecom-SudParis.eu

Steve Jeffrey Tueno Fotso
stuenofotso@gmail.com

1 SAMOVAR, CNRS UMR 5157, Télécom SudParis, Institut Polytechnique de Paris, Evry, France

2 GRIF, Département d’informatique, Faculté des sciences, Université de Sherbrooke, Sherbrooke, QC, Canada

3 Université Paris-Est, LACL, UPEC, IUT Sénart Fontainebleau, Fontainebleau, France

[1] www.event-b.org.

[2] www3.hhu.de/stups/prob.

aspects/objectives of the system that involved the first and the third authors, the first author wrote an initial Event- B specification which is then discussed with all the other members, especially about the adequate structure for representing the TDDs and the VSS. Once the data structure of the track has been decided, the first author continued the design of the Event- B model without interacting with the rest of the team except to discuss the invariant violations with respect to the ambiguous/incorrect description of the VSS state changes. The invariant violations have been pointed out by running ProB and also by playing the scenarios provided in the specification document. The second author has participated to this validation activity. The development of the model provided in [7] took about one month and half, and its updating according to the new specification document release took one more month.

Regarding the paper published at the ABZ'18 conference [8], the present article introduces an updated Event- B model according to the revised standard provided in [4]. This new release corrected some of the inconsistencies pointed out by the previous modelling approaches by updating mainly the behaviour of the timers and also the description of the transitions. Hence, the present paper brings the following additional contributions/updates:

- a connected train sends information to the controller each time its position changes: enters into the track, moves to the next TTD/VSS, etc,
- a ERTMS train is allowed to enter into the track only if a movement authority, which includes the first VSS of the track, is assigned to it.
- more transitions are modelled according to the new release,
- some inconsistencies pointed out in the previous modelling were not applicable any more, but new ones have been detected in the new release,
- as more transitions have been modelled, more scenarios have been validated with the new model release.

This paper is structured as follows. In Sect. 2, we provide a short introduction to Event- B and ProB. Our modelling conventions are described in Sect. 3. In Sect. 4, we describe our modelling strategy, explaining how we take into account controller and environment characteristics. Section 5 provides an overview of our Event- B model; it describes the refinement strategy, explaining the order in which various features of the ERTMS were taken into account. In Sect. 6, we present each refinement. In Sect. 7, we discuss how the requirements and our specification of it have been verified. We provide in Sect. 8 a brief comparison of our model with others presented at ABZ2018. We conclude in Sect. 9 with an appraisal of this work. In the sequel, we suppose that the reader can access to the case study text, in order to avoid unnecessary repetitions.

2 An overview of Event-B and ProB

2.1 Event-B

Event- B [1] was designed to support the incremental construction of system specifications using stepwise refinement. It is complementary to the B method [9], which can be used to refine a system specification into an implementation. Event- B and B share the same mathematical notation, but use different refinement relations. Event- B refinement allows for the introduction of new events and strengthening of event guards, whereas B only allows for data refinement and precondition weakening.

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 to a more concrete one. An Event- B specification is made of two elements: *context* and *machine*. A context describes the static part of an Event- B specification; it consists of constants and sets (user-defined types) together with axioms 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 should hold are specified using an invariant, denoted Inv, written using a first-order predicate on the state variables:

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

An event has the following form:

ANY
X
WHEN
G
THEN
Act
END

An event is said to be enabled when its guard G holds. A machine transition consists in the execution of a single event, among all enabled events. Action Act of an event describes the modifications made to the state variables. In this paper, we restrict ourselves to actions of the form $\mathbf{x} := \mathbf{e}$.

The execution of an event should maintain the invariant. To this aim, proof obligations are generated. For each event, we have to establish that:

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

The expression $[Act]Inv$ denotes the actions Act applied as a substitution to formula Inv; it denotes the weakest constraint on the *before* state such that the execution of Act leads to an *after* state satisfying Inv.

Refinement is a process of enriching or modifying a model in order to augment the functionality being modelled, or/and explain how some purposes are achieved. Contexts and machines 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. New events of a machine M' that refines M are considered to refine the **skip** event of M; hence, they cannot modify a variable introduced in M. Consequently, all events that need to modify a variable v are introduced where v is first declared. In this paper, the refined events have the same form as abstract events:

ANY
X_r
WHEN
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 should 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 refined action should be stronger than the effect of the abstract one:

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

To discharge the different proof obligations, the Rodin platform offers an automated prover; other provers can also be added as plug-ins, like SMT provers VERIT [10] and CVC3 [11], and the provers of Atelier B,[3] the main platform supporting the B method, which we use in this work. The Rodin and Atelier B provers offer an automatic and an interactive mode to discharge the proof obligations.

2.2 PROB

PROB is an animator and explicit automatic model checker, originally developed for the verification and validation of software development based on the B language; it now also supports several other languages (TLA, CSP, ALLOY, Z). Developed at the University of Düsseldorf since 2003, PROB implements an automatic model checking technique to check LTL (linear temporal logic) [12] and CTL (Computational Tree Logic) [13] properties against a B specification. The core of PROB is written in Prologue; its purpose is to be a comprehensive tool in the area of formal verification methods. Its main functionalities can be summarised up as follows:

1. PROB can find a sequence of operations which, starting from a valid initial state of the machine, leads to a state that violates its invariant,
2. giving a valid state, PROB can exhibit the operation that make the invariant violated,
3. PROB allows the animation of a B/EventB specification to allow the user to play different scenarios from a given starting state that satisfies the invariant. Through a graphical user interface implemented in Tcl/Tk, the animator provides the user with: (i) the current state, (2) the history of the operation executions that has led to the current state and (3) a list of all the enabled operations, along with proper argument instantiations. In this way, the user does not have to guess the right values for the operation arguments.
4. PROB supports the model checking of LTL and CTL assertions.

[3] www.atelierb.eu.

3 Modelling conventions

We reuse the terminology introduced in [14]. A control system interacts with its environment using sensors and actuators. A sensor measures the value of some environment characteristic m, called a *monitored* variable (e.g., train on a track), and provides this measure (e.g., detection of an object on the track) to the software controller as an *input* variable i. In a perfect world, we have $m = i$, but a sensor may fail. 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.

4 Requirements and modelling strategy

The modelling, presented in the current paper, is based on the new version of the standard [4], released in July 2018 (version 1C), after the ABZ'2018 conference. Note that [2] is based on version 1A released in July 2017 [3].

The system consists of a train supervisor called the *trackside* which is responsible for setting trackside signals and also to communicate with trains. There are two types of trains: ERTMS trains and non-ERTMS trains. ERTMS trains can communicate with the train supervisor to report their position and to receive a movement authority (MA), which describes how far they can move. Non-ERTMS trains cannot communicate with the supervisor; they are detected (but not identified) by trackside sensors, and they obey trackside signals to determine how far they can go. In addition, some ERTMS trains are equipped with an onboard train integrity monitoring system (TIMS) that can report the train integrity to the supervisor. A train is said to be *integer* if all of its cars are linked to the head of the train; a train is non-integer when some of its cars are unlinked. This gives three types of trains: (i) TIMS-ERTMS, (ii) ERTMS without TIMS and (iii) non-ERTMS trains. We assume that trains move on one single track, all in the same direction. We also take into account trains that can enter and move on this track without reporting their position to the supervisor (i.e., non-ERTMS trains or disconnected trains).

The track is divided into sections called TTD (Trackside Train Detection). A TTD is equipped with sensors that can detect the presence of an object, which can be a train, or something else; it cannot identify a train with this sensor. A TTD is further divided into subsections called VSS (virtual subsection). The TIMS can be used to determine the VSS occupied by the train and the train's integrity. A train can lose its integrity by splitting into several parts.

The supervisor periodically computes a MA and sends it to ERTMS Trains. A MA specifies the VSS that the train can move up to, but never beyond, in order to avoid collision with another train ahead. As stated in the case study, the computation of MAs is out of scope; we simply nondeterministically choose a MA that avoids a collision with the trains ahead. Trains can be connected or disconnected with the supervisor. When connected, a train reports its position and integrity to the supervisor on a regular basis. Timers are used to detect disconnected trains and to manage ghost trains. A ghost train is either a physical object or a non-ERTMS train present on the track and detected by a TTD, but for which no position report has been received, and a failure of the TTD sensors which incorrectly report the presence of an nonexistent object.

In a first step, we model train occupation and train location in terms of TTDs. Occupation refers to the actual physical position of a train, while location refers to the position known by the supervisor. Occupation and location do not always coincide (e.g., unconnected train or non-ERTMS Train). Trains move freely without constraints; sensors report the status of a TTD and trains report their location to the controller. In a second step, we refine TTDs in terms of VSS occupation and location. Finally, we add the timers and the movement authority since some VSS states depend on them. In the real world, the cycle duration of the controller is adapted such that it can deal with any change in the environment (train movement, timer expiration, etc.). To accurately model such a behaviour, we give priority to controller events over other events modelling the train movement and timers. Moreover, as the EVENT- B language does not support timing aspects, the values of the timers are not modelled; for us a timer is only seen as active or not and as running or not. The coding of the VSS state transition and the timers is achieved according to their descriptions provided in the standard in a modular manner, that is, a modelling element is associated with each of them. It makes it easier to trace back an error detected in the verification phase to the standard. In addition to calculating the VSS states, we model and verify a safety property that specifies that two connected trains cannot occupy the same VSS. In other words, this property ensures that ERTMS trains do not collide. Table 1 summarises up the requirements of the system we dealt with.

5 Model overview

Our model contains three contexts. Context C0 declares constants related to the track. We consider a single track which is represented by an interval of natural numbers $minTTD..maxTTD$ (**Req 1**). A stronger typing using an

Table 1 Summary of the ERTMS requirements

Req. No.	Description
Req 1	The train track is partitioned into several fixed TTD sections
Req 2	The system should accommodate three types of trains: TIMS-equipped ERTMS, ERTMS not fitted with TIMS and non-ERTMS
Req 3	Each TTD is partitioned into one or more fixed VSSs
Req 4	VSS can have one of the following statuses: *free*, *occupied*, *ambiguous*, or *unknown*
Req 5	A TTD can be reported as *free* or *occupied*
Req 6	A TTD is reported as free if and only if there are no trains or a part of a train located on the TTD.
Req 7	The trains travel along a straight line track and in the same direction
Req 8	For ERTMS trains, their EoAs are defined in terms of the VSSes
Req 9	For non-ERTMS trains, their EoAs are defined in terms of TTD sections
Req 10	The MAs are disjoint, i.e., trains will be safe from collision if they respect the provided MAs
Req 11	A mute timer is assigned to each train
Req 12	A wait integrity timer is assigned to each train
Req 13	A disconnected propagation timer is assigned to each VSS
Req 14	A ghost train propagation timer is assigned to each TTD
Req 15	ERTMS trains should not collide

abstract set TTD would be more type safe, but it makes the proofs more cumbersome, as we have experienced in the first drafts of our specification. This is why each TTD is represented by a natural number of this interval. TTDs are ordered using their number. The set of trains is partitioned into trains or cars (i.e., cars that have accidentally split from a train). Constant *trainKind* indicates for each actual train whether it is a TIMS train (*TimErtms*), an ERTMS train without TIMS (*Ertms*) or a non-ERTMS train (*NoErtms*) (**Req 2**) . Only TIMS and ERTMS trains can connect and send their information to the supervisor.

CONTEXT C0
SETS
TRAINS StateTTD TrainKind
CONSTANTS
freeT occupiedT Ttds minTTD maxTTD trainKind
TimErtms Ertms NoErtms Trains Cars
AXIOMS
axm1 : $finite(TRAINS)$
axm2 : $partition(StateTTD, \{freeT\}, \{occupiedT\})$
axm3 : $partition(TRAINS, Trains, Cars)$
axm4 : $minTTD \in \mathbb{N}_1 \wedge maxTTD \in \mathbb{N}_1 \wedge$
$minTTD \leq maxTTD$
axm7 : $Ttds = minTTD \,..\, maxTTD$
axm8 : $partition(TrainKind, \{TimErtms\},$
$\{Ertms\}, \{NoErtms\})$
axm9 : $trainKind \in Trains \rightarrow TrainKind$

Context C1 declares the VSSs, which are also modelled as an interval of naturals. We use a total, monotonic, surjective function *TtdOfVss* to associate a VSS with its TTD (**Req 3**).

axm4 : $Vss = minVSS \,..\, maxVSS$
axm5 : $TtdOfVss \in Vss \twoheadrightarrow Ttds$
axm6 : $\forall v_1, v_2 \cdot \{v_1, v_2\} \subseteq Vss \wedge v_1 < v_2 \Rightarrow$
$TtdOfVss(v_1) \leq TtdOfVss(v_2)$

Context C2 declares an abstract set *StateVSS* = {*freeV*, *occupiedV*, *unknown*, *ambiguous*} to represent the states of a VSS from the supervisor viewpoint (**Req 4**). A VSS in state *freeV* contains no train. A VSS in state *occupiedV* contains a single train. State *unknown* denotes a VSS for which it is unknown whether there is a train on it. State *ambiguous* denotes a VSS which contains at least one train; it is not sure whether there is more than one train.

The specification is structured into four refinement steps (i.e., five machines). Machine M0 introduces the trains, the supervisor and the unsupervised movements of trains on TTDs. Machine M1 introduces the reporting of positions by trains to the supervisor, but still without supervision of their movement. Machine M2 introduces the VSS, still without supervision. Machine M3 introduces movement supervision with MAs and the computation of VSS states using timers and other information. A final refinement M4 is introduced to prove the main safety property, namely that trains do not collide when following MAs. Table 2 relates the components of our model with these requirements and those listed in [2].

Table 2 Cross-reference between the components of our model, the requirements of Table 1 and the requirements and assumptions of [2]

Component	Reqs of Table 1	R & A of [2]
C0	1, 2	2, 4, 11
C1	3	3
C2	4	6
M0	5, 6, 7	1, 5, 6
M1		
M2		
M3	8 . . . 14	7 . . . 10, 12 . . . 36
M4	15	

6 Refinements

In this section, we briefly describe each refinement. The complete archive of the Event- B project is available in [8].

6.1 Machine M0: free movement on TTDs

This machine contains five variables. Controller variable *stateTTD* faithfully represents the real state of TTDs (**Req 5** and **Req 6**) (i.e., the case study assumes $m = i$ for this variable). Environment variables *trainOccupationTTDRear* and *trainOccupationTTDFront*, respectively, denote the first and last TTD occupied by a given train. Environment variable *isConnected* denotes whether a train is connected to the supervisor. This variable denotes a total function including the trains that are not on track because some of them should be connected to receive the authorisation to enter on the track. Boolean variable *supervisor* is used to guard train movements to ensure that other events like train supervision are interleaved with train movements. The following invariants type these variables. Symbols "$\rightarrow$" and "$\mathrel{+\!\!\!\!\rightarrow}$", respectively, denote a total function and a partial function.

inv1 : $stateTTD \in Ttds \rightarrow StateTTD$
inv2 : $trainOccupationTTDFront \in TRAINS \mathrel{+\!\!\!\!\rightarrow} Ttds$
inv3 : $trainOccupationTTDRear \in dom(trainOccupationTTDFront) \rightarrow Ttds$
inv4 : $\forall tr \cdot tr \in dom(trainOccupationTTDFront) \Rightarrow trainOccupationTTDRear(tr) \leq trainOccupationTTDFront(tr)$
inv5 : $isConnected \in trainKind^{-1}[\{Ertms, TimErtms\}] \rightarrow BOOL$
inv6 : $supervisor \in BOOL$

The set of trains on the track is represented by the domain of the function *trainOccupationTTDFront* (i.e., $dom(trainOccupationTTDFront)$). We consider events that model the sensing of all the TTD states by the supervisor, the entering and exiting of a train on the track, the movement of a train on the track, the connection and disconnection of a train and also the sending of a position by a connected train. The movement of a train is decomposed into three events to distinguish between the cases where the train moves within the same TTD, the front of the train enters a new TTD and the rear of the train leaves a TTD (**Reqs 6, 7**). This decomposes the proofs for train movement into smaller ones. Trains move freely, and collisions can occur at this level. The supervisor does not know the position of a train; it only knows the states of TTDs. Also, we have defined an event to split a train into two parts, the train with the engine and the cars left behind, to model the loss of integrity. As a simple illustration, we provide below the specification of event trainSupervisor.

Event trainSupervisor $\widehat{=}$
any $ttds$ $active$
where
grd1 : $supervisor = TRUE$
grd2 : $ttds = (\bigcup tr \cdot tr \in dom(trainOccupationTTDFront) \mid trainOccupationTTDRear(tr)..trainOccupationTTDFront(tr))$
grd3 : $active \in BOOL$
then
act1 : $stateTTD := (ttds \times \{occupiedT\}) \cup ((Ttds \setminus ttds) \times \{freeT\})$
act2 : $supervisor := active$
end

Guard grd1 checks whether it is the turn of the supervisor to run. This guard aims at avoiding that the supervisor runs indefinitely, for simulation and validation purposes. Guard grd2 constrains event local variable *ttds* to the set of TTDS which are occupied by trains. Action act1 updates TTD states. Action act2 nondeterministically gives controls to either the trains or the supervisor using the choice made in guard grd3.

We also specify an event to model the expiration of a timer. At this level, it only sets the variable *supervisor* to *TRUE*. We specify this event in the first machine since it has to modify the variable *supervisor* to give control to the supervisor to process the received information. Indeed, any event introduced in next refinements cannot update the variables defined before.

6.2 Machine M1: Trains reporting their positions

This machine adds the controller variables *trainLocationTTDRear* and *trainLocationTTDFront* to store train positions as reported by ERTMS trains. The case study assumes that reports are accurate. The following invariants provide the types of these variables. Note that the location of a train on a track may be unknown to the supervisor. Thus, *trainLocationTTDFront* is modelled as a partial function of the domain of *trainOccupationTTDFront*, which denotes the real train position. Invariant inv3 states that the rear is known only for TIMS-ERTMS trains that have already provided their front positions.

inv1 : $trainLocationTTDFront \in dom(trainKind^{-1}[\{TimsErtms, Ertms\}] \lhd trainOccupationTTDFront) \mathrel{+\!\!\!\!\rightarrow} Ttds$

$$
\begin{aligned}
&\texttt{inv2} : trainLocationTTDRear \in \\
&\quad dom(trainLocationTTDFront) \mathrel{+\!\!\!\rightarrow} Ttds \\
&\texttt{inv3} : trainKind^{-1}[\{TimErtms\}] \cap \\
&\quad dom(trainLocationTTDFront) \\
&\qquad = dom(trainLocationTTDRear) \\
&\texttt{inv4} : \forall tr \cdot tr \in dom(trainLocationTTDFront) \Rightarrow \\
&\quad trainLocationTTDRear(tr) \leq \\
&\qquad trainLocationTTDFront(tr)
\end{aligned}
$$

To take these new invariants into account, the events modelling train movements and train connection of a disconnected train are refined (extended) to report the position of an ERTMS train. New actions are added to these events to report the position of a train by modifying controller variables *trainLocationTTDRear* and *trainLocationTTDFront* using the two environment variables *trainOccupationTTDRear* and *trainOccupationTTDFront*. Train integrity is nondeterministically chosen to reflect the possibility of loosing it at any point. When train integrity is lost, the rear position of a train is not updated, in order to ensure that its last known rear position remains and to avoid collision with the preceding train when computing the MA. However, there is no provision in M1 to avoid collision; this is introduced in M3. The following action is added to the event modelling the update of the rear position. Indeed, the rear of train is updated only for connected TIMS trains with a confirmed integrity:

$$
\begin{aligned}
&\texttt{act}: trainLocationTTDRear := \\
&\quad \{ \\
&\qquad TRUE \mapsto \\
&\qquad\quad trainLocationTTDRear \mathbin{\lhd\!\!\!+} \{tr \mapsto newRear\}, \\
&\qquad FALSE \mapsto trainLocationTTDRear \\
&\quad \}(\\
&\qquad bool(\\
&\qquad\quad tr \mapsto TRUE \in isConnected\ \wedge \\
&\qquad\quad trainKind(tr) = TimErtms\ \wedge \\
&\qquad\quad integ = TRUE) \\
&\qquad)
\end{aligned}
$$

Action `act` uses an expression that simulates a conditional expression of the form

$$\textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2,$$

which does not exist in Event- B. It is instead written in the form $\{TRUE \mapsto e_1, FALSE \mapsto e_2\}(bool(c))$, which evaluates the function $\{TRUE \mapsto e_1, FALSE \mapsto e_2\}$ at condition c.

6.3 Machine M2: Introducing VSSs

Recall that a TTD is divided into VSSs. This data refinement refines the train position variables based on TTDs (i.e., *trainOccupationTTDx* and *trainLocationTTDx*) with position variables based on VSSs. New environment variables *trainOccupationVSSRear* and *trainOccupationVSSFront* represent the real VSS position of a train. New controller variables *trainLocationVSSRear* and *trainLocationVSSFront* represent the VSS positions computed by the supervisor using train reports.

$$
\begin{aligned}
&\texttt{inv5} : trainLocationVSSFront \in \\
&\quad dom(trainLocationTTDFront) \rightarrow Vss \\
&\texttt{inv6} : trainLocationVSSRear \in \\
&\quad dom(trainLocationVSSFront) \rightarrow Vss \\
&\texttt{inv7} : trainOccupationVSSFront \in \\
&\quad dom(trainOccupationTTDFront) \rightarrow Vss \\
&\texttt{inv8} : trainOccupationVSSRear \in \\
&\quad dom(trainOccupationVSSFront) \rightarrow Vss
\end{aligned}
$$

Four gluing invariants stating that the VSS positions and the TTD positions are consistent, for both the controller and the environment, using function *TtdOfVss*, are also added, like the following one.

$$
\begin{aligned}
&\texttt{inv11} : \forall tr \cdot tr \in dom(trainOccupationVSSFront) \Rightarrow \\
&\quad TtdOfVss(trainOccupationVSSFront(tr)) \\
&\qquad = trainOccupationTTDFront(tr)
\end{aligned}
$$

No new event is added. The existing events are refined to take into account these new variables. As in M1, train collisions can occur in M2.

6.4 Machine M_3: Computing VSS states and assigning MAs

6.4.1 Introducing new variables

This refinement is the most complex one. The state of each VSS is computed, and MAs are assigned to trains. At this level, the integrity and the length information of a train are stored by two Boolean variables since they are used in the VSS computation. Timers are introduced to detect disconnected trains, the loss of integrity and ghost trains.

As stated before, the management of the timers, MAs and the calculation of VSS states have been updated in the last release of the standard [4]. So, the modelling of these concepts is quite different from that presented in [8]. For instance, the MA for ERTMS trains is defined in terms of VSS, while the MA of non-ERTMS trains is defined in terms of TTD. The following three invariants are defined for the new variables introduced to model the MA of an ERTMS train (**Req 8**). Controller variables *MATrainRearVSS* and *MATrainFrontVSS* define the MA of each ERTMS train. A MA is an interval of VSSs. Invariant `inv7` in Fig. 1 states

Fig. 1 Disjointness of MAs in machine M_3

$$
\begin{array}{ll}
\texttt{inv1} : & MATrainRearVSS \in (Trains - trainKind^{-1}[\{NoErtms\}]) \nrightarrow Vss \\
\texttt{inv2} : & MATrainFrontVSS \in dom(MATrainRearVSS) \to Vss \\
\texttt{inv3} : & \forall tr \cdot tr \in dom(MATrainRearVSS) \Rightarrow MATrainRearVSS(tr) \leq MATrainFrontVSS(tr) \\
\texttt{inv4} : & MATrainRearTTD \in trainKind^{-1}[\{NoErtms\}] \nrightarrow TTD \\
\texttt{inv5} : & MATrainFrontTTD \in dom(MATrainRearTTD) \to TTD \\
\texttt{inv6} : & \forall tr \cdot tr \in dom(MATrainRearTTD) \Rightarrow MATrainRearTTD(tr) \leq MATrainFrontTTD(tr) \\
\texttt{inv7} : & \forall tr1, tr2 \cdot \\
& \quad tr1 \neq tr2 \wedge \\
& \quad \{tr1, tr2\} \subseteq dom(MATrainFrontTTD \cup MATrainFrontVSS) \\
& \Rightarrow \\
& \qquad \{TRUE \mapsto min(TtdOfVss^{-1}[\{MATrainRearTTD(tr1)\}]), \\
& \qquad FALSE \mapsto MATrainRearVSS(tr1)\} \\
& \qquad (bool(trainKind(tr1) = NoErtms)) \\
& \quad .. \\
& \qquad \{TRUE \mapsto max(TtdOfVss^{-1}[\{MATrainFrontTTD(tr1)\}]), \\
& \qquad FALSE \mapsto MATrainFrontVSS(tr1)\} \\
& \qquad (bool(trainKind(tr1) = NoErtms)) \\
& \quad \cap \\
& \qquad \{TRUE \mapsto min(TtdOfVss^{-1}[\{MATrainRearTTD(tr2)\}]), \\
& \qquad FALSE \mapsto MATrainRearVSS(tr2)\} \\
& \qquad (bool(trainKind(tr2) = NoErtms)) \\
& \quad .. \\
& \qquad \{TRUE \mapsto max(TtdOfVss^{-1}[\{MATrainFrontTTD(tr2)\}]), \\
& \qquad FALSE \mapsto MATrainFrontVSS(tr2)\} \\
& \qquad (bool(trainKind(tr2) = NoErtms)) \\
& = \varnothing
\end{array}
$$

Fig. 2 Disjointness of MAs in machine M_3 (a simplified version)

$$
\begin{array}{ll}
\texttt{inv1} : & MATrainRear \in Trains \nrightarrow Vss \\
\texttt{inv2} : & MATrainFront \in dom(MATrainRear) \to Vss \\
\texttt{inv3} : & \forall tr \cdot tr \in dom(MATrainRear) \Rightarrow MATrainRear(tr) \leq MATrainFront(tr) \\
\texttt{inv4} : & \forall tr.\ tr \in Trains \wedge trainKind(tr) = NoErtms \Rightarrow \\
& \qquad (\exists\, td1, td2.(td1 \in Ttds \wedge td2 \in Ttds \wedge td1 \leq td2 \wedge \\
& \qquad\quad MATrainRear(tr) = min(TtdOfVss^{-1}[\{td1\}]) \wedge \\
& \qquad\qquad MATrainFront(tr) = max(TtdOfVss^{-1}[\{td2\}]))) \\
\texttt{inv5} : & \forall tr1, tr2. \\
& \qquad tr1 \in dom(MATrainFront) \wedge \\
& \qquad tr2 \in dom(MATrainFront) \wedge \\
& \qquad tr1 \neq tr2 \\
& \quad \Rightarrow \\
& \qquad MATrainRear(tr1)..MATrainFront(tr1) \\
& \qquad \cap \\
& \qquad MATrainRear(tr2)..MATrainFront(tr2) = \varnothing
\end{array}
$$

that the MAs of two different trains are disjoint, to avoid collisions (**Req 10**). It uses a conditional expression on the train type: ERTMS trains have a MA based on VSS; non-ERTMS trains have a MA based on TTD, and thus, we must convert this TTD MA into a VSS MA, starting with the smallest VSS of the TTD for the start position and the greatest VSS of the TTD for the end position (**Req 9**).

As one can remark, invariant inv7 makes the specification more complicated, especially for the expression of the guards. We decided to simplify this by defining only two variables *MATrainFrontVSS* and *MATrainFrontVSS* as described in Fig. 2. Invariant inv4 ensures that non-ERTMS trains have a MA that starts with the first VSS of a TTD and ends with the last VSS of a TTD. Invariant inv5 states the disjointness of MAs.

The next invariants introduce timers: timers related to trains may be *running* or *expired* while those associated with the VSS and TTD may be *running* or *expired* but also *inactive* (the three values are grouped in an enumerated set *TimerValues*). Contrary to our previous modelling introduced in [8], mute and integrity timers are associated with each train even when a train is not located on the track yet. The following invariants model the timers managed by the system as described by the requirements **Req 11-Req 14**.

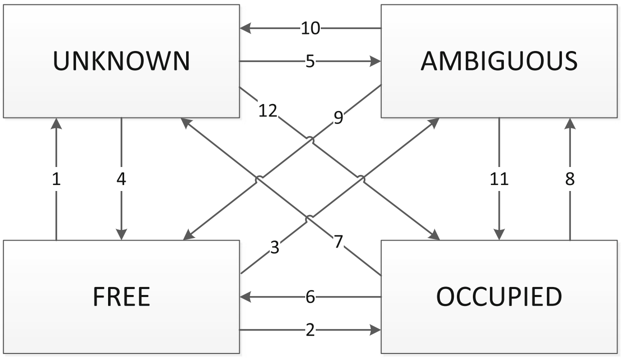


Fig. 3 State machine of VSS reproduced from Fig. 7 of [4]

$$\begin{aligned}
&\texttt{inv5} : muteTimer \in \\
&\quad trainKind^{-1}[\{TimErtms, Ertms\}] \rightarrow \\
&\quad\quad \{running, expired\} \\
&\texttt{inv6} : integrityTimer \in \\
&\quad trainKind^{-1}[\{TimErtms, Ertms\}] \rightarrow \\
&\quad\quad \{running, expired\} \\
&\texttt{inv7} : disconnectTimer \in Vss \rightarrow TimerValues \\
&\texttt{inv8} : ghostTimer \in Ttds \rightarrow TimerValues
\end{aligned}$$

The *muteTimer* is used to detect that a train has failed to report its position within the required time frame; in that case, the state of the VSSs in front of that train and within the train's MA becomes *unknown*.

Finally, the following variables are introduced to compute the VSS states.

$$\begin{aligned}
&\texttt{inv9} : currentStateVSS \in Vss \rightarrow StateVSS \\
&\texttt{inv10} : previousFront \in \\
&\quad dom(trainLocationVSSFront) \twoheadrightarrow Vss \\
&\texttt{inv11} : previousFrontState \in \\
&\quad dom(previousFront) \twoheadrightarrow StateVSS
\end{aligned}$$

Variables *previousFrontState* and *previousFront*, respectively, record the previous value of *currentStateVSS* and the previous front position of the trains. They are, respectively, updated when the supervisor computes the states of the VSS and when the train reports its position; they are needed in the computation of some VSS state transitions.

6.4.2 Modelling VSS state machine transitions

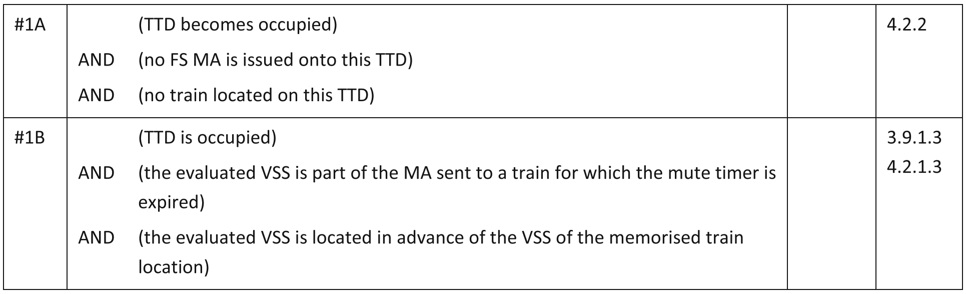
The main complexity of this refinement is to compute the VSS states, which depend on several conditions. These conditions are described by a state machine in Figure 7 of [4] and reproduced here in Fig. 3. The guards of its transitions are described, using natural language, in Table 2 of [4]. This table spans 3 pages (pp. 28–30). Figure 4 provides an excerpt of this table. The guard of a transition i in Fig. 3 is given by the disjunction of the guards labelled #iX in Fig. 4. For example, the guard of transition 1 is #1A ∨ #1B ∨ ... ∨ #1F; only #1A and #1B are shown in Fig. 4. Some transitions have priority over others (e.g., #2A has precedence over #3A).

Ideally, the computation of the state of each VSS should be done in a single event, because the states must be all computed before assigning MAs. It also ensures that Table 2 of [4] is deterministic, i.e., well-defined. Furthermore, it allows for taking into account the priority between transitions for a given VSS. We have coded the state machine of Fig. 3 into a single event, namely trainSupervisor. We use guard numbers (e.g., #1A) to name local variables of the event (e.g., *vss1A*). Such a variable is constrained to contain the new state values for the VSSs satisfying the corresponding guard. For instance, set *vss1A* contains the VSSs satisfying guard #1A and their state will change from FREE to UNKNOWN. The union of sets *vssiX* is used to update state variable *currentStateVSS* in event trainSupervisor.

To illustrate our approach, we provided in Fig. 5 an excerpt of the guards of event trainSupervisor that models guards #1A and #1B of Fig. 4. Guard grd4 of Fig. 5 represents guard #1A. We use a quantified union to identify the VSSs satisfying #1A. It reads as follows: a VSS must currently be free, since transition 1 starts from state FREE; it must also be on a TTD that *becomes* occupied (first conjunct of guard #1A) and any VSS of this TTD must not be within a MA or occupied by a train (second and third conjuncts of #1A). Variable *ttds* denotes the occupied TTDs in the after state, whereas *stateTTD* denotes the before value, which allows us to determine that a TTD becomes occupied. The resulting state of these VSSs is UNKNOWN as given by transition 1, which is represented by taking the Cartesian product of the VSSs returned by the quantified union with the singleton set $\{unknown\}$. In summary, guard #1A says that the TTD sensor detected an object, but the supervisor has no record of a train on a VSS of that TTD; thus, its status is unknown.

Several ambiguities have been found. For instance, in the description of transition #2, the third condition states that "VSS where the train was located before it was located on the evaluated VSS, was occupied" can be interpreted as an implication, which means that the guard holds even when the train does have a previous location. Or, it can be interpreted as a conjunction, which means that the train should have a previous position, for the guard to hold. For this specific transition, we choose the second semantics as the first one increases the nondeterminism with the other transitions. Moreover, as we are not experts of the domain, we decided not to elicit further these aspects, especially that similar expressions with different semantics are used in other transitions, like the transition #6.

Fig. 4 Excerpt of Table 2 in [4]



#1A	 AND AND	(TTD becomes occupied) (no FS MA is issued onto this TTD) (no train located on this TTD)		4.2.2
#1B	 AND AND	(TTD is occupied) (the evaluated VSS is part of the MA sent to a train for which the mute timer is expired) (the evaluated VSS is located in advance of the VSS of the memorised train location)		3.9.1.3 4.2.1.3

$$
\begin{array}{l}
\texttt{grd4} : vss1A = \\
\quad (\bigcup vs \cdot \\
\qquad currentStateVSS(vs) = freeV \land \\
\qquad vs \in TtdOfVss^{-1}[ttds - stateTTD^{-1}[\{occupiedT\}]] \land \\
\qquad (\forall tr \cdot tr \in dom(MATrainFront) \implies TtdOfVss(vs) \notin MATrainRear(tr) \mathinner{..} MATrainFront(tr)) \land \\
\qquad (\forall tr \cdot tr \in dom(trainLocationVSSRear) \implies TtdOfVss(vs) \notin \\
\qquad\qquad\qquad TtdOfVss(trainLocationVSSRear(tr))..TtdOfVss(trainLocationVSSFront(tr))) \\
\quad\ \mid \{vs\}) \times \{unknown\} \\
\texttt{grd5} : vss1B = \\
\quad (\bigcup vs \cdot \\
\qquad currentStateVSS(vs) = freeV \land \\
\qquad vs \in TtdOfVss^{-1}[stateTTD^{-1}[occupiedT]] \land \\
\qquad (\exists tr \cdot tr \in dom(muteTimer) \land muteTimer(tr) = FALSE \land \\
\qquad\quad vs \in MATrainRear(tr) \mathinner{..} MATrainFront(tr) \land vs > trainLocationVSSFront(tr)) \\
\quad\ \mid \{vs\}) \times \{unknown\}
\end{array}
$$

Fig. 5 Excerpt of the guards of event trainSupervisor corresponding to Fig. 4

7 Requirements verification and model validation

This section describes the verifications carried out using the provers of Rodin (EVENT- B's development platform) and the model checker/animator PROB [15] plug-in for Rodin. Our strategy to verify the development and the requirements is as follows. We used PROB mainly to discover possible invariant violations prior to the proof phase that may be long and complex. PROB has proved to be a useful and effective tool to check the sequencing of the events. We have also used it to play the scenarios provided in the case study to validate our specification.

7.1 Proving the determinacy of the VSS state machine

Recall that state variable *currentStateVSS* is typed as a function. The proof obligation generated by this typing invariant ensures that each VSS state has a single new value; hence, there is a single transition that updates it. This is equivalent to prove that the VSS state machine described in the case study is deterministic. This turns out to be fairly complex. For each VSS state value (e.g., FREE), there are three outgoing transitions to the other three possible VSS state values (e.g., transitions 1, 2 and 3 of Fig. 3). To ensure determinacy, we must prove that the guards of these three transitions are mutually disjoint. Let n_i be the number of disjuncts in the disjunctive normal form of the guard of transition i. Then, we have to consider $n_i * n_j$ cases in the proof of disjointness of transitions i and j. Luckily, transitions priorities eliminate a few cases to consider. In total, there are **47** high-level cases to consider, which is a significant proof effort.

One way to simplify the handling of this proof in Rodin would be to decompose event trainSupervisor into four events, one for each VSS state value. That would still allow us to prove the determinacy of the VSS state machine, but we would lose the atomicity of VSS state computation. We would then have to control the ordering of events to ensure that these four events are computed before assigning MAs. For the sake of simplicity and to ease the construction of the overall specification, we have chosen to use a single event.

7.1.1 Using PROB to check the determinacy of the VSS state machine

PROB can be used to find invariant violations with counterexamples. We have used this feature extensively. The counterexamples provided help in identifying the missing guards and invariants required to prove invariant preservation. However, the state space of machine M3 is huge, with its 23 variables, most of them typed as functions. PROB will only check the reachable states, and when it does not terminate in a reasonable time, one cannot determine which interesting conditions have been explored, for instance, among the 47 cases of guard disjointness discussed earlier.

An alternative way to check the determinacy of the VSS state machine is to use the constraint satisfier of ProB, which can find models for a formula. ProB uses it to find values of constants in an Event- B context. To specifically check one case among the 47 cases for the determinacy of the VSS state machine, we construct a new context that declares the state variables, used in the guards of the VSS state machine, as constants, and their related invariants as axioms. We finally add to this context the local variables of event trainSupervisor that computes new sets of VSS states and we check that these two sets are not disjoint (e.g., check that $dom(vss1A) \cap dom(vss2A) \neq \{\}$). If ProB finds a model for this context, it means that the corresponding transition guards in the VSS state machine are not disjoint, given the invariants used in our machine. It thus means that the invariants are insufficient to prove the determinacy of the VSS state machine and that they must be enriched or strengthened.

7.1.2 Dealing with inconsistencies of the VSS state machine

We have found several cases where the guards are not disjoint, which means that one of the following three alternatives holds: (i) our representation of the guards are incorrect, and (ii) the case study text is incorrect, (iii) invariants are missing to rule out these counterexamples (i.e., these Event- B states are not reachable from the initial state of the system). Since we are not experts of the ERTMS standard, it is hard for us to determine which alternative holds. In a first model of the system, ProB finds several counterexamples when searching for invariant violations that lead to a state where two transitions are not disjoint. Such traces are due, for instance, to the expiration of several timers reported at the same moment as the reporting of the train position. Thus, we do not know if the case study is wrong, or if this trace is impossible in the real world where the timers represent actual clocks with different values or perhaps there are implicit assumptions in the case study that we missed or we could not figure out by simply reading it. To rule them out, we assume that the transitions depending on the timers are dealt with last; priority is given to those depending on the train position. From the Event-B point of view, we use the overload operator to express it. Moreover, as for the representation of the guards, we have used a straight forward translation of the phrasal terms of the natural language text into state variables, to simplify as much as possible the translation of the guards. However, there is still the possibility of misinterpreting the natural language text. The term of "train location" is not very precise in the standard: according to our first understanding, it corresponds to the information sent by the train, but later at page 27, it is stated that such an information is updated by the TTD information without explaining how this is done. Another point that is important is related to the conditions under which the trains are allowed to enter into the track. At the beginning, since no explicit condition is given, we thought that a train cannot enter into the track without having a MA that includes the first VSS, but the first step of Scenario 4 contradicts such an assumption. In addition, at the end of the document, some scenarios are provided in order to validate and illustrate the update of the VSS state. We think that more details should be provided at each step to understand how the transitions are executed. It would be interesting to show how the state of the VSS changes immediately when a train goes on a next VSS. This is not clear on the provided figures that may imply that the train jumps from a VSS to the next one. Moreover, the scenarios use some terms that are not defined in the document such as *end of mission*. Also, it is not clear which kinds of trains (ERTMS, TIMS, No ERTMS) are considered in the scenarios. Given the length of the case study, our limited expertise in the domain and the number of ambiguities or missing (implicit) assumptions, we decided not to elicit further these aspects, because there is no point in making hypothetical (as opposed to "realistic") assumptions in order to prove the determinacy of the VSS state machine. The key issue is more to be able to identify ambiguities, thanks to formalisation, validation and verification. In a real context, they can be resolved in a systematic manner using domain experts. Moreover, since proof obligations can be independently discharged, not proving the determinacy of the state machine does not prevent us from proving the main safety property; we can assume that the VSS state machine can be made deterministic. In addition, the four VSS states can be reduced to only two (free or occupied), since the other two are used to manage potentially hazardous situations, as noted in the case study (paragraph 3.2.1.1.1 of [4]). From a refinement viewpoint, it would have been easier to start with a simple state machine that contains only two states, prove its determinacy and the absence of collision at this level and then further refine into a state machine with four states.

7.2 Proving safety properties

We have stated one main safety property, which is that two ERTMS trains cannot be on the same VSS, and thus, ERTMS trains should not collide, but non-ERTMS trains could (**Req 15**). This property is expressed in Fig. 6 using the environment variables *trainOccupationVSSRear* and *trainOccupationVSSFront*, which represent the real position of the trains (not the position as known by the supervisor). This proof was conducted in a new refinement machine M4, for the sake of modularity. The guards of events that modify these variables are based solely on the controller variables and thus represent the fact that trains move according to their MAs computed by the supervisor. If the invariant holds, it means that trains following their MAs should not collide. To prove this property, we needed to add and prove the auxiliary invariants presented in Fig. 7, which can be seen as

Fig. 6 Invariant representing the absence of collision for ERTMS trains

$$
\begin{array}{l}
\texttt{inv1} : \forall tr1, tr2 \cdot \\
\quad tr1 \in Trains \land tr2 \in Trains \land \\
\quad tr1 \neq tr2 \land \\
\quad tr1 \in dom(trainOccupationVSSFront) \land \\
\quad tr2 \in dom(trainOccupationVSSFront) \land \\
\quad trainKind(tr1) \in \{TimErtms, Ertms\} \land \\
\quad trainKind(tr2) \in \{TimErtms, Ertms\} \\
\Rightarrow \\
\qquad\quad trainOccupationVSSRear(tr1) \mathinner{..} trainOccupationVSSFront(tr1) \\
\qquad \cap\, trainOccupationVSSRear(tr2) \mathinner{..} trainOccupationVSSFront(tr2) \\
\quad = \varnothing
\end{array}
$$

Fig. 7 Auxiliary invariants for proving the absence of collision for ERTMS trains

$$
\begin{array}{l}
\texttt{inv2} : \forall tr \cdot \\
\quad tr \in Train \land \\
\quad tr \in dom(trainOccupationVSSFront) \land \\
\quad trainKind(tr) \in \{TimErtms, Ertms\} \\
\Rightarrow \\
\quad tr \in dom(MATrainFront) \land \\
\quad trainOccupationVSSRear(tr) \mathinner{..} trainOccupationVSSFront(tr) \subseteq \\
\qquad MATrainRear(tr) \mathinner{..} MATrainFront(tr) \\
\texttt{inv3} : \forall tr \cdot \\
\quad tr \in dom(trainOccupationVSSFront) \land \\
\quad tr \in dom(trainLocationVSSRear) \\
\Rightarrow \\
\quad trainOccupationVSSRear(tr) \geq trainLocationVSSRear(tr) \\
\texttt{inv4} : \forall tr \cdot \\
\quad tr \in dom(trainOccupationVSSFront) \land \\
\quad tr \in dom(trainLocationVSSRear) \\
\Rightarrow \\
\quad trainOccupationVSSFront(tr) \geq trainLocationVSSFront(tr)
\end{array}
$$

Table 3 Rodin proof statistics of the case study

Component	PO	Auto	Manual	Reviewed	Undischarged
C0	0	0	0	0	0
C1	3	1	2	0	0
C2	0	0	0	0	0
Animation	0	0	0	0	0
M0	34	23	11	0	0
M1	60	16	44	0	0
M2	132	27	105	0	0
M3	125	26	91	0	8
M4	36	9	27	0	0

lemmas required for the main proof. Invariant inv2 states that a ERTMS train can occupy only the VSS included in its MA. Invariants inv3 and inv4 state that the position of a train known by the supervisor is behind the real position of the train. Recall that the case study assumes that the position reported by trains is accurate.

Table 3 provides the proof statistics of the case study. It is worth noting that the correctness of the proof carried out on the machine $M4$ depends on the correctness of the 8 proof obligations that remain to discharge in machine $M3$. These proof obligations are related to the determinacy of the VSS state machine, and they are hard to discharge, as we model the VSS state changes in a single very complex event. The problem can be resolved by splitting this event into several ones, one per transition of the VSS state machine. In that case, the execution of each event would correspond to the firing of a unique transition of the VSS state machine. With this approach, the safety property would still hold, because the invariants used to prove the property would still be valid. However, such a model is not faithful to the real system, because it would not ensure the determinacy of the VSS state machine. Proving the determinacy ensures that a single transition can be fired in a given state.

8 Comparison with other models

This section reports on the work that dealt with the same case study and presented at the ABZ2018 conference. Different formal languages and verification approaches have been used for that purpose. In [16], the system is modelled in PROMELA, and the verification is performed using

Table 4 Comparison of different approaches

Criteria	Work					
	[16]	[17]	[18]	[19]	[20]	[21,22]
Graphical modelling	–	–	–	–	iUML- B	SysML-Kaos
Formal Language	PROMELA	B language	Electrum	Event- B	Event- B	Event- B
Verification method	Model checking/SPIN	ProB+Proofs	Electrum analyser	Proofs	Proofs	Proofs
Non-collision property	Violated	–	–	Assumed	–	–
Inconsistency report	+	+	+	–	–	–

the SPIN model checker. The authors make some restrictions by considering, for instance, that a train can occupy at most two VSSs. Moreover, the state of each VSS is calculated independently of others and the behaviour of the timers is nondeterministic. Such restrictions produce a less realistic and generic model, especially that only two trains are considered as assumed in the scenarios of the standard. In [17], the specification of the system is written in the B notation, and the validation and the verification is performed using the provers of AtelierB and ProB. Even if the update of the VSS state is done in a loop, this loop is executed once on each VSS, but the standard provides that the final value of a VSS state may require several additional loops in order to take the new VSS states on which it can depend. Also, no detail is given about the modelling of the timers.

An Electrum specification of the system is provided in [18]. Electrum is an extension of the Alloy language with events that specifies a system in terms, among others, of signatures for data types and assertions on state transitions. The Electrum analyser can check the existence of a model that satisfies the specification and check properties about the specification. The Electrum analyser also allows one to explore the state transitions and traces of the specification. LTL formulas can be used to construct the specification, as well as to prove properties about a specification. Electrum supports bounded model checking using SAT solvers and unbounded model checking using SMV. Electrum does not allow for refinement.

An Event- B specification, very similar approach to ours, is presented in [19]. However, the assumptions used to simplify the system make the resulting model more restrictive. Indeed, only the mute timer is considered, but the other kinds of timers also affect the VSS state. Moreover, assuming that a train can only move to a free VSS is too strong and should be in fact the safety property to demonstrate. An iUML- B approach for modelling the case study is also developed in [20]. The approach uses UML class and state machine diagrams to model the topology of the trackside and the dynamic aspects of the system. These diagrams are then automatically translated into Event- B using the iUML- B plug-in. The authors do not give any detail about the possible inconsistencies they detect during the proof activity. Moreover, the safety property they defined does not ensure the non-collision of two trains as they assume that two MA trains may share a common VSS. Let us note that most work also identified several inconsistencies and ambiguities in the specification of the VSS transitions.

In a companion paper, we explore the use of ontologies and SysML/KAOS to model this case study [21,22]. The approach is goal refinement oriented and starts with a global property as the root goal of the SysML/KAOS goal model, which is refined into more elementary goals using different AND and OR refinement patterns. The data of the system are specified by ontologies. The translation of these diagrams into Event- B, according to rules defined in [23], permits to produce a complete formal specification that can be validated and verified under the Rodin tool. A more detailed comparison of these approaches is provided in this companion paper.

Table 4 summarises up the characteristics of the approaches described above according to a set of criteria. Symbols "–" and "+ " indicate whether a characteristic is taken into account ("+ ") or not ("–").

Let us recall that, in this paper, we do not deal with the actual values of the timers. Indeed, we only model their states as active or not, without giving them any value. Thus, we think that some counterexamples are due to this restriction since a timer can expire at any moment even immediately after it was launched. In the literature, some very suitable models have been developed for the modelling of the timing constraints like Alur and Dill's timed [24] or Dutertre and Sorea's calendar automata [25]. A real-time Maude model of ERTMS level 2 is proposed in [26]; discrete time is used and model-checked using the real-time Maude tool. Note that ERTMS level 2 is a different standard from the hybrid ERTMS level 3.

Several approaches have been proposed to enrich the Event- B language by introducing time constraints [27–29]. We do not explore these approaches since our initial objective is to experience the plain Event- B method on the case study and to draw some conclusions on its applicability even if we know in advance that no discrete neither real time is directly supported in Event- B, especially that the description of the case study does not provide the values to which the timers are set. Finally, it would be interesting to look deeper into the existing structuring mechanisms proposed for EVENT- B: decomposition [30] or modularisation [31], in order to structure the specification into logical units to permit operation calls like in the B method.

9 Conclusion

Our model covers the essential parts of the case study as described in [4]. By making some additional assumptions on the movement of the ERTMS trains for example, we were able to prove the safety of ERTMS trains. It only remains to prove the determinacy of the VSS state machine, which could not be completed because of the ambiguities of the case study text. Understanding the case study itself was a challenge, because of the difficulty to identify missing assumptions. Determining the ordering of events was anything but trivial. Domain experts typically write for other domain experts; it is not natural for them to think of all the details that a non-expert does not know.

We have found EVENT- B to be adequate to model this case study even if some constraints of this method make the obtained model more complex. For instance, the event that manages the timers has been introduced in the first model because a new event cannot modify existing variables. Also, we modelled the movement of a train with three distinct events in order to reduce the complexity of the proof activity by generating more proof obligations but with a low complexity. The use of the PROB model checker turned out to be very useful for an early error detection by simulating different scenarios. In this paper, we deliberately chose not to use any EVENT- B plug-ins (e.g., [32,33]) in order to be able to compare our solution with solutions based on them (and assuming that a paper using them would be submitted to ABZ2018). Moreover, we think that the PROB plug-in of the Rodin platform should be improved to allow users to replay a stored scenario without being forced to do it step by step, especially when scenarios are very long.

Acknowledgements This research was supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) and the FORMOSE project funded by the French National Research Agency (ANR).

References

1. Abrial, J.: Modeling in Event-B. Cambridge University Press, Cambridge (2010)
2. Hoang, T.S., Butler, M.J., Reichl, K.: The hybrid ERTMS/ETCS level 3 case study. In: Butler, M.J., Raschke, A., Hoang, T.S., Reichl, K. (eds.): Abstract State Machines, Alloy, B, TLA, VDM, and Z—6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings. Volume 10817 of Lecture Notes in Computer Science, pp. 251–261. Springer, Berlin (2018)
3. EEIG ERTMS Users Group: Hybrid ERTMS/ETCS Level 3: Principles, Ref. 16E042 Version 1A. Technical report, Brussels, Belgium (2017)
4. EEIG ERTMS Users Group: Hybrid ERTMS/ETCS Level 3: Principles, Ref. 16E042 Version 1C. Technical report, Brussels, Belgium (2018)
5. Voisin, L., Abrial, J.: The Rodin platform has turned ten. In: Ameur, Y.A., Schewe, K. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z—4th International Conference, ABZ 2014, Toulouse, France, June 2–6, 2014. Proceedings. Volume 8477 of Lecture Notes in Computer Science, pp. 1–8. Springer, Berlin (2014)
6. Leuschel, M., Butler, M.J.: Prob: an automated analysis toolset for the B method. STTT **10**(2), 185–203 (2008)
7. Mammar, A., Frappier, M., Fotso, S.J.T., Laleau, R.: An Event-B model of the hybrid ERTMS/ETCS level 3 standard. In: Butler, M.J., Raschke, A., Hoang, T.S., Reichl, K. (eds.): Abstract State Machines, Alloy, B, TLA, VDM, and Z—6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings. Volume 10817 of Lecture Notes in Computer Science, pp. 353–366. Springer, Berlin (2018)
8. Mammar, A., Frappier, M., Fotso, S.J.T., Laleau, R.: An Event-B Model of the Hybrid ERTMS/ETCS Level 3 Standard. http://info.usherbrooke.ca/mfrappier/abz2018-ERTMS-Case-Study (2018). Accessed Jan 2018
9. Abrial, J.: The B-Book—Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
10. Bouton, T., Oliveira, D.C.B.D., Déharbe, D., Fontaine, P.: veriT: an open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) Automated Deduction—CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2–7, 2009. Proceedings. Volume 5663 of Lecture Notes in Computer Science, pp. 151–156. Springer, Berlin (2009)
11. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3–7, 2007, Proceedings. Volume 4590 of Lecture Notes in Computer Science, pp. 298–302. Springer (2007)
12. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October–1 November 1977, pp. 46–57. IEEE Computer Society (1977)
13. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking–History, Achievements, Perspectives. Volume 5000 of Lecture Notes in Computer Science, pp. 196–215. Springer, Berlin (2008)
14. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Program. **25**(1), 41–61 (1995)
15. Leuschel, M., Butler, M.J.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003: Formal Methods Europe, Pisa, Italy, September 8–14, 2003. Volume 2805 of LNCS, pp. 855–874. Springer, Berlin (2003)
16. Arcaini, P., Jezek, P., Kofron, J.: Modelling the hybrid ERTMS/ETCS level 3 case study in spin. In: Butler, M.J., Raschke, A., Hoang, T.S., Reichl, K. (eds.): Abstract State Machines, Alloy, B, TLA, VDM, and Z—6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings. Volume 10817 of Lecture Notes in Computer Science, pp. 277–291. Springer, Berlin (2018)
17. Hansen, D., Leuschel, M., Schneider, D., Krings, S., Körner, P., Naulin, T., Nayeri, N., Skowron, F.: Using a formal B model at runtime in a demonstration of the ETCS hybrid level 3 concept with real trains. In: Butler, M.J., Raschke, A., Hoang, T.S., Reichl, K. (eds.): Abstract State Machines, Alloy, B, TLA, VDM, and Z—6th

International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings. Volume 10817 of Lecture Notes in Computer Science, pp. 292–306. Springer, Berlin (2018)

18. Cunha, A., Macedo, N.: Validating the hybrid ERTMS/ETCS level 3 concept with electrum. In: Butler, M.J., Raschke, A., Hoang, T.S., Reichl, K. (eds.): Abstract State Machines, Alloy, B, TLA, VDM, and Z—6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings. Volume 10817 of Lecture Notes in Computer Science, pp. 307–321. Springer, Berlin (2018)
19. Abrial, J.: The ABZ-2018 case study with event-b. In: Butler, M.J., Raschke, A., Hoang, T.S., Reichl, K. (eds.): Abstract State Machines, Alloy, B, TLA, VDM, and Z—6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings. Volume 10817 of Lecture Notes in Computer Science, pp. 322–337. Springer, Berlin (2018)
20. Dghaym, D., Poppleton, M., Snook, C.F.: Diagram-led formal modelling using iUML-b for hybrid ERTMS level 3. In: Butler, M.J., Raschke, A., Hoang, T.S., Reichl, K. (eds.): Abstract State Machines, Alloy, B, TLA, VDM, and Z—6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings. Volume 10817 of Lecture Notes in Computer Science, pp. 338–352. Springer, Berlin (2018)
21. Fotso, S.J.T., Frappier, M., Laleau, R., Mammar, A.: Modeling the hybrid ERTMS/ETCS level 3 standard using a formal requirements engineering approach. In: Butler, M.J., Raschke, A., Hoang, T.S., Reichl, K. (eds.): Abstract State Machines, Alloy, B, TLA, VDM, and Z—6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings. Volume 10817 of Lecture Notes in Computer Science, pp. 262–276. Springer, Berlin (2018)
22. Fotso, S.J.T., Frappier, M., Laleau, R., Mammar, A.: Modeling the Hybrid ERTMS/ETCS Level 3 Implementation through Goal Diagrams and Ontologies Using the FORMOSE Approach. http://info.usherbrooke.ca/mfrappier/abz2018-ERTMS-Case-Study-Formose (2018). Accessed Jan 2018
23. Fotso, S.J.T., Mammar, A., Laleau, R., Frappier, M.: Event-B expression and verification of translation rules between SysML/KAOS domain models and B system specifications. In: Butler, M.J., Raschke, A., Hoang, T.S., Reichl, K. (eds.): Abstract State Machines, Alloy, B, TLA, VDM, and Z—6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings. Volume 10817 of Lecture Notes in Computer Science, pp. 55–70. Springer, Berlin (2018)
24. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
25. Dutertre, B., Sorea, M.: Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In: FORMATS/FTRTFT. Volume 3253 of Lecture Notes in Computer Science, pp. 199–214. Springer, Berlin (2004)
26. Berger, U., James, P., Lawrence, A., Roggenbach, M., Seisenberger, M.: Verification of the european rail traffic management system in real-time maude. Sci. Comput. Program. **154**, 61–88 (2018)
27. Cansell, D., Méry, D., Rehm, J.: Time constraint patterns for Event-B development. In: B. Volume 4355 of Lecture Notes in Computer Science, pp. 140–154. Springer, Berlin (2007)
28. Sarshogh, M.R., Butler, M.J.: Specification and refinement of discrete timing properties in Event-B. ECEASST **46**, 1–15 (2011)
29. Mammar, A., Laleau, R.: Modeling a landing gear system in Event-B. STTT **19**(2), 167–186 (2017)
30. Silva, R., Pascal, C., Hoang, T.S., Butler, M.J.: Decomposition tool for Event-B. Softw. Pract. Exp. **41**(2), 199–208 (2011)
31. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A.B., Varpaaniemi, K., Ilic, D., Latvala, T.: Supporting reuse in Event-B development: Modularisation approach. In: ASM Volume 5977 of Lecture Notes in Computer Science, pp. 174–188. Springer, Berlin (2010)
32. Fathabadi, A.S., Butler, M.J., Rezazadeh, A.: Language and tool support for event refinement structures In Event-B. Formal Asp. Comput. **27**(3), 499–523 (2015)
33. Said, M.Y., Butler, M.J., Snook, C.F.: A method of refinement in UML-B. Softw. Syst. Model. **14**(4), 1557–1580 (2015)

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