An Event-B Model of the Hybrid ERTMS/ETCS Level 3 Standard

Amel Mammar[1], Marc Frappier[2], Steve Jeffrey Tueno Fotso[2,3](✉), and Régine Laleau[3]

[1] Télécom SudParis, SAMOVAR-CNRS, Évry, France
amel.mammar@telecom-sudparis.eu

[2] Laboratoire GRIL, Département d'informatique, Faculté des sciences, Université de Sherbrooke, Québec, Canada
marc.frappier@usherbrooke.ca

[3] LACL, Université Paris-Est Créteil, Créteil, France
steve.tuenofotso@univ-paris-est.fr, laleau@u-pec.fr

Abstract. This paper presents an Event-B model of the ABZ2018 case study on the European Rail Traffic Management System (ERTMS) standard. The case study focusses on the management of fixed virtual subsections (VSS). We model the hybrid level 3 of the standard, which assumes that trains may be either equipped with an on-board train integrity monitoring system (TIMS) and that they report their position and integrity, ERTMS trains not fitted with TIMS that report only their front position or non-ERTMS trains that do not report any information about their position. We take into account most of the main features of the case study. 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.

Keywords: Hybrid ERTMS/ETCS level 3 · Event-B · ProB
Control system

1 Introduction

This paper proposes an Event-B model of the hybrid ERTMS/ETCS level 3 case study [5] proposed for ABZ2018. The case study concerns the European Rail Traffic Management System (ERTMS), the system of standards for management and interoperation of signalling for railways by the European Union. For the sake of concision, we only provide a brief overview of the case study. The reader is referred to [2] for more details.

This paper is structured as follows. In Sect. 2, we summarize the characteristics of the standard that we have taken into account in our model. In Sect. 3, we

M. Butler et al. (Eds.): ABZ 2018, LNCS 10817, pp. 353–366, 2018.
https://doi.org/10.1007/978-3-319-91271-4_24

describe our modelling strategy, explaining how we take into account controller and environment characteristics, while in Sect. 4, we present an overview of our Event-B model. We describe the refinement strategy, explaining the order in which the various features of the ERTMS were taken into account. In Sect. 5, we describe each refinement. In Sect. 6, we discuss how the requirements and our specification of it have been verified. We conclude in Sect. 7 with an appraisal of this work. In the sequel, we suppose that the reader can read the case study text, in order to avoid unnecessary repetitions.

2 Modelled Characteristics

We model the hybrid level 3 of the standard, which assumes that trains may be equipped with an on-board train integrity monitoring system (TIMS) and that they report their position and integrity to the train supervisor (the system controller, called the *trackside* in the case study), ERTMS trains not fitted with TIMS that report only their front position or non-ERTMS trains that do not report any information about their position. We assume that trains move on a single track, all in the same direction. We also take into account trains that can enter and move on the track without reporting their position to the supervisor (*i.e.*, non-ERTMS trains).

A 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 sub-section). 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 an MA (Movement Authority) and sends it to ERTMS Trains. An 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 an 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 that is present on the track and detected by a TTD, but for which no position report has been received, or a failure of the TTD sensors which incorrectly report the presence of an inexistant object.

3 Modelling Conventions

We reuse the terminology introduced in [8]. 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, call 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 Model Overview

Event-B models are iteratively constructed using refinement. A model component can either be a context or a machine. A context contains constants declaration. A machine contains events that modify state variables. A machine can refine another machine; a context can extends another context. A machine can see contexts to have access to its constants. Each refinement adds new information to the model; these could be new state variables, data refinement of state variables, new events or new properties. Event-B refinement [1] allows for guard strengthening, non-determinism reduction, and new events introduction. 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.

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 $minTTD \mathinner{..} maxTTD$. A stronger typing using an 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, a ERTMS train or a non-ERTMS train. 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 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 : $Ttds = minTTD \mathinner{..} maxTTD$

$$\texttt{axm8} : partition(TrainKind, \{TimErtms\}, \{Ertms\}, \{NoErtms\})$$
$$\texttt{axm9} : trainKind \in Trains \rightarrow TrainKind$$

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 to its TTD.

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

Context C2 declares an abstract set $StateVSS = \{freeV, occupiedV, unknown, ambiguous\}$ to represent the states of a VSS from the supervisor view point. 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 are more than one train.

The specification is structured into four refinement steps (*i.e.*, four 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 informations. A final refinement M4 is introduced to prove the main safety property, namely that trains do not collide when following MAs.

5 Refinements

In this section, we briefly describe each refinement. The complete archive of the Event-B project is available in [7].

5.1 Machine M0: Free Movement on TTDs

This machine contains five variables. Controller variable *stateTTD* faithfully represents the real state of TTDs (*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 authorization to enter on the track. Boolean variable *trainMvt* 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 “→” and “⇸” respectively denote a total function and a partial function.

inv1 : $stateTTD \in Ttds \rightarrow StateTTD$

inv2 : $trainOccupationTTDFront \in TRAINS \nrightarrow Ttds$

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 : $trainMvt \in BOOL$

The set of trains on the track is represented by the domain of 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. 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. 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 $ttds \quad active$

where

grd1 : $ttds = (\bigcup tr \cdot tr \in dom(trainOccupationTTDFront) \mid$
$trainOccupationTTDRear(tr) \mathinner{..} trainOccupationTTDFront(tr))$

grd2 : $active \in BOOL$

then

act1 : $stateTTD := (ttds \times \{occupiedT\}) \cup ((Ttds \setminus ttds) \times \{freeT\})$

act2 : $trainMvt := active$

end

Guard grd1 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 grd2.

5.2 Machine M1: Trains Reporting Their Positions

This machine adds controller variables *trainLocationTTDRear* and *trainLocationTTDFront* to store in the supervisor 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 modeled 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 : $trainLocationTTDFront \in dom(Trains \lhd trainOccupationTTDFront) \nrightarrow Ttds$
inv2 : $trainLocationTTDRear \in dom(trainLocationTTDFront) \nrightarrow Ttds$
inv3 : $trainKind^{-1}[\{TimErtms\}] \cap dom(trainLocationTTDFront) \subseteq dom(trainLocationTTDRear)$
inv4 : $\forall tr \cdot tr \in dom(trainLocationTTDFront) \Rightarrow trainLocationTTDRear(tr) \leq trainLocationTTDFront(tr)$

This refinement introduces a new event, trainSnd, to report train positions. Existing events are refined (extended) to take into account the new variables. Event trainSnd reports the position of a train by modifying controller variables *trainLocationTTDRear* and *trainLocationTTDFront* using the environment variables *trainOccupationTTDRear* and *trainOccupationTTDFront*. Train integrity is non-deterministically 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 specification of event trainSnd is provided below. Action act2 simulates an if-then-else by using a set containing two tuples of the form $\{TRUE \mapsto e_1, FALSE \mapsto e_2\}$; hence this set is a function and it is evaluated with the value of *integ*, acting like **if** *integ* **then** e_1 **else** e_2. Guard grd6 ensures that the reported position does not decrease, since a train cannot move backward.

Event trainSnd $\widehat{=}$
  any tr $integ$ $lengch$
  where
    grd2 : $tr \in dom(trainOccupationTTDFront)$
    grd3 : $tr \in dom(isConnected) \wedge isConnected(tr) = TRUE$
    grd4 : $integ \in BOOL$
    grd5 : $tr \in trainKind^{-1}[\{TimErtms\}] \wedge tr \notin dom(trainLocationTTDRear) \Rightarrow integ = TRUE$
    grd6 : $tr \in dom(trainLocationTTDFront) \Rightarrow trainOccupationTTDFront(tr) \geq trainLocationTTDFront(tr)$
  then
    act2 : $trainLocationTTDRear := \{TRUE \mapsto trainLocationTTDRear \mathbin{\lhd\!\!\!+} \{tr \mapsto trainOccupationTTDRear(tr)\}, FALSE \mapsto trainLocationTTDRear\}\ (integ)$
    act3 : $trainLocationTTDFront(tr) := trainOccupationTTDFront(tr)$
  end

5.3 Machine M2: Introducing VSSs

Recall that a TTD is divided into VSSs. This refinement data replaces (*i.e.*, data 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.

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

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.

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

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.

5.4 Machine M_3: Computing VSS States and Assigning MAs

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 loose of integrity and ghost trains. New variables are introduced and typed using the following invariants.

$$\begin{aligned}
&\texttt{inv1} : MATrainRear \in dom(trainLocationVSSFront) \nrightarrow Vss\\
&\texttt{inv2} : MATrainFront \in dom(MATrainRear) \rightarrow Vss\\
&\texttt{inv3} : \forall tr \cdot tr \in dom(MATrainRear) \Rightarrow\\
&\qquad\qquad MATrainRear(tr) \leq MATrainFront(tr)\\
&\texttt{inv4} : \forall tr1, tr2 \cdot tr1 \in dom(MATrainFront) \land\\
&\qquad tr2 \in dom(MATrainFront) \land tr1 \neq tr2 \Rightarrow\\
&\qquad\quad MATrainRear(tr1) \mathinner{..} MATrainFront(tr1)\\
&\qquad \cap\, MATrainRear(tr2) \mathinner{..} MATrainFront(tr2)\\
&\qquad = \varnothing
\end{aligned}$$

Controller variables *MATrainRear* and *MATrainFront* define the MA of each train that the supervisor knows. An MA is an interval of VSSs. Invariant inv4 states that the MAs of trains are disjoint, to avoid collisions.

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*.

inv5 : $muteTimer \in dom(trainLocationVSSFront) \rightarrow \{running, expired\}$
inv6 : $integrityTimer \in dom(trainLocationVSSFront) \rightarrow \{running, expired\}$
inv7 : $disconnectTimer \in Vss \rightarrow \{inactive, running, expired\}$
inv8 : $ghostTimer \in Ttds \rightarrow \{inactive, running, expired\}$

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.

inv9 : $currentStateVSS \in Vss \rightarrow StateVSS$
inv10 : $previousFront \in dom(trainLocationVSSFront) \nrightarrow Vss$
inv11 : $previousFrontState \in dom(previousFront) \nrightarrow StateVSS$

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.

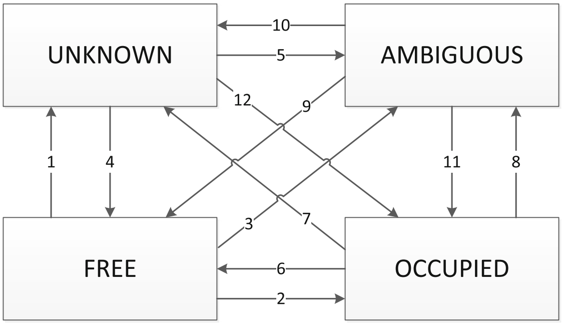
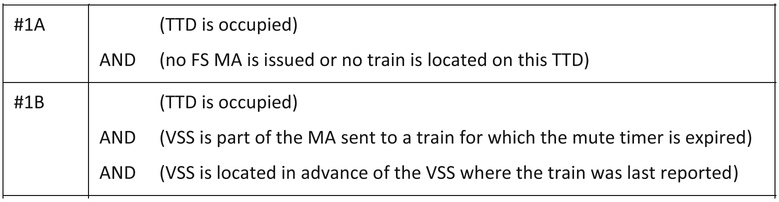


Fig. 1. The state machine of VSS reproduced from Fig. 7 of [2]

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 Fig. 7 of [2] and reproduced here in Fig. 1. The guards of its transitions are described, using natural language, in Table 2 of [2]. This table spans 3.5 pages (pp. 24–28). Figure 2 provides an excerpt of this table. The guard of a transition i in Fig. 1 is given by the disjunction of the guards labeled `#`iX in Fig. 2. For example, the guard of transition 1 is `#1A` $\vee$ `#1B` $\vee \ldots$ `#1F`; only `#1A` and `#1B` are shown in Fig. 2. Some transitions have priority over others (*e.g.*, guards `#2A` and `#2B` have precedence over transition 3).

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 [2] 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. 1 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 *vss iX* is used to update state variable *currentStateVSS* in event trainSupervisor.

To illustrate our approach, we provide in Fig. 3 an excerpt of the guards of event trainSupervisor that models guards #1A and #1B of Fig. 2. Guard grd4 of Fig. 3 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 start from state FREE; it must also be on an occupied TTD (first conjunct of guard #1A) and any VSS of this TTD must not be within an MA or occupied by a train (second conjunct of #1A). 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 it's status is unknown.



#1A		(TTD is occupied)
	AND	(no FS MA is issued or no train is located on this TTD)
#1B		(TTD is occupied)
	AND	(VSS is part of the MA sent to a train for which the mute timer is expired)
	AND	(VSS is located in advance of the VSS where the train was last reported)

Fig. 2. An excerpt of Table 2 in [2]

6 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 [6] plug-in for Rodin. ProB is an explicit state-based model checker for the B methods (classic B and Event-B) and several others (TLA, CSP, Alloy). 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.

$$\begin{array}{l}
\texttt{grd4} : vss1A = \\
\qquad (\bigcup vs \cdot currentStateVSS(vs) = freeV \quad \land \\
\qquad\quad vs \in ttds \quad \land \\
\qquad\quad ((\forall tr \cdot tr \in dom(MATrainFront) \Rightarrow TtdOfVss(vs) \notin \\
\qquad\qquad TtdOfVss(MATrainRear(tr)) \,..\, \\
\qquad\qquad TtdOfVss(MATrainFront(tr))) \quad \lor \\
\qquad\quad\; (\forall tr \cdot tr \in dom(trainLocationVSSFront) \Rightarrow TtdOfVss(vs) \notin \\
\qquad\qquad TtdOfVss(trainLocationVSSRear(tr)) \,..\, \\
\qquad\qquad TtdOfVss(trainLocationVSSFront(tr)))) \\
\qquad \mid \{vs\}) \times \{unknown\} \\
\texttt{grd5} : vss1B = \\
\qquad (\bigcup vs \cdot currentStateVSS(vs) = freeV \quad \land \\
\qquad\quad vs \in ttds \quad \land \\
\qquad\quad (\exists tr \cdot (tr \in dom(muteTimer) \quad \land \\
\qquad\qquad muteTimer(tr) = FALSE \quad \land \\
\qquad\qquad vs \in MATrainRear(tr) \,..\, MATrainFront(tr)) \quad \land \\
\qquad\qquad vs \geq trainLocationVSSFront(tr)) \\
\qquad \mid \{vs\}) \times \{unknown\}
\end{array}$$

Fig. 3. An excerpt of the guards of trainSupervisor corresponding to Fig. 2

6.1 Proving Safety Properties

We have stated one main safety property, which is that two TIMS/ERTMS trains cannot be on the same VSS, and thus TIMS/ERTMS trains should not collide, but non-ERTMS trains could. This property is expressed 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.

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

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 following invariants, which can be seen as lemmas required for the main proof.

inv2 : $\forall tr \cdot tr \in dom(trainOccupationVSSFront) \quad \land$
$tr \in Train \land trainKind(tr) \in \{TimErtms, Ertms\} \Rightarrow$
$tr \in dom(MATrainFront) \land$
$trainOccupationVSSRear(tr) \mathbin{..} trainOccupationVSSFront(tr)$
$\subseteq$
$MATrainRear(tr) \mathbin{..} MATrainFront(tr)$

inv3 : $\forall tr \cdot tr \in dom(trainOccupationVSSFront) \land$
$tr \in dom(trainLocationVSSRear) \Rightarrow$
$trainOccupationVSSRear(tr) \geq trainLocationVSSRear(tr)$

inv4 : $\forall tr \cdot tr \in dom(trainOccupationVSSFront) \land$
$tr \in dom(trainLocationVSSRear) \Rightarrow$
$trainOccupationVSSFront(tr) \geq trainLocationVSSFront(tr)$

Invariant inv2 tates that a TIMS/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 are accurate.

6.2 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 proving 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. 1). 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 disjonctive 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.

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 counter-examples 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 22 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 enriched or strengthened.

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, (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 expert 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 leads 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. For instance, consider the following conjunct of #2A.

#2A		...
	AND	(VSS where the estimated front end of the train was last reported, was "occupied" after the processing of this previous position report)

This conjunct can be interpreted as an implication, which means that the guard holds even when only one position report has been issued for the train. Or, it can be interpreted as a conjunction, which means that at least two position reports must have been issued for the train, for the guard to hold. 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 resolve 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 [2]).

7 Conclusion

Our model covers the essential parts of the case study. We were able to prove the safety of TIMS/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. In this paper, we deliberately chose not to use any Event-B plugins (*e.g.*, [3,9]) in order to be able to compare our solution with solutions based on them (and assuming that a paper using them will be submitted to ABZ2018). In a companion paper, we explore the use of ontologies and SysML/KAOS to model this case study [4].

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. EEIG Ertms Users Group: Hybrid ERTMS/ETCS Level 3: Principles. Technical report, Brussels, Belgium, July 2007
3. 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)

4. 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, February 2018. http://info.usherbrooke.ca/mfrappier/abz2018-ERTMS-Case-Study-Formose
5. Hoang, T.S., Butler, M., Reichl, K.: The hybrid ERTMS/ETCS level 3 case study. Technical report, ECS, University of Southampton, U.K. (2007)
6. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
7. Mammar, A., Frappier, M., Fotso, S.J.T., Laleau, R.: An Event-B Model of the Hybrid ERTMS/ETCS Level 3 Standard, February 2018. http://info.usherbrooke.ca/mfrappier/abz2018-ERTMS-Case-Study
8. Parnas, D.L., Madey, J.: Functional documents for computer systems. Sci. Comput. Program. **25**(1), 41–61 (1995)
9. Said, M.Y., Butler, M.J., Snook, C.F.: A method of refinement in UML-B. Softw. Syst. Model. **14**(4), 1557–1580 (2015)