Modelling the Hybrid ERTMS/ETCS Level 3 Case Study in SPIN*

Paolo Arcaini, Pavel Ježek, and Jan Kofroň

Charles University, Faculty of Mathematics and Physics, Czech Republic
{arcaini,jezek,kofron}@d3s.mff.cuni.cz

Abstract. The SPIN model checker has been successfully applied to the modelling, validation, and verification of different safety-critical systems. In this paper, we model and validate the Hybrid ERTMS/ETCS Level 3 Case Study using SPIN; in particular, we show the assumptions we made to keep the state space limited, and present the problems and ambiguities that arose during the modelling. Although SPIN offers several advantages in terms of validation and verification facilities, its modelling language PROMELA is limited if compared to higher level notations of other formal methods. Therefore, we discuss the advantages and disadvantages of using the tool, and how it could be improved in terms of modelling facilities.

1 Introduction

In the context of the ABZ 2018 conference, the Hybrid ERTMS/ETCS Level 3 Case Study [6] has been proposed as benchmark for comparing the strengths (and weaknesses) of different state-based formal methods. The solutions provided for the case study should demonstrate the modelling, validation, and verification facilities of different methods.

The aim of the Hybrid ERTMS/ETCS Level 3 [6] is to increase the throughput of railway tracks, by integrating the physical information coming from the *trackside detection system* with information transmitted by the train itself regarding its position and integrity. In pre-Level 3 systems the railway track is divided in TTD (*trackside train detection*) sections and entering and exiting each TTD is physically detected; in such a situation, a whole TTD section is blocked when there is a train inside (i.e., no two trains can be in a single TTD[1]). In Hybrid ERTMS/ETCS Level 3 the train also periodically sends information about its position and integrity to the trackside system; in this situation, each TTD is further divided into several *virtual sub-sections* (VSSs) and the system should avoid presence of two trains on the same VSS. We remind to [6] for the complete description of the requirements.

* The research reported in this paper has been partially supported by the Czech Science Foundation project number 17-12465S.

[1] Actually, two trains can be in a TTD if they are operating in *on-sight mode* in which the drivers are fully responsible for the train movement; this setting, however, is an exceptional case that is not a part of normal operational mode.

As widely demonstrated in literature [14], there is no golden method for system analysis, and each particular method can provide a better support for some aspects (e.g., modelling), but be deficient on some others (e.g., scalability in verification). High-level notations as Abstract State Machines [4] and B [1] provide a wide support for modelling and refining the model and can be also used for documentation purposes when discussing with the stakeholders. In addition, they also provide different facilities for verification in terms of model checking [2,15]; they usually translate their models into the notation of an existing model checker [2,7,17,16] and use this for the verification. The problem of this approach is that the mapping usually introduces a non-trivial overhead that limits their scalability.

On the other hand, implementing the problem directly in the notation provided by the model checker (e.g., PROMELA for SPIN [12] or the input notation of NuSMV [8]) usually allows one to obtain a more simple model that scales better. This is due to the fact that the notation provided by model checkers is rather limited and allows to get a better understanding of the consumed resources. However, such notations are usually less readable than notations as ASMs and B. Therefore, there is a trade-off between readability and scalability.

In this paper, we propose a solution of the aforementioned case study in SPIN. In developing the system, we tried to abstract as much as possible from all the unnecessary details, but still preserving the soundness of the verification. Since we have taken the approach of modelling the system directly in the input language of an explicit model checker, our solution can be taken as a baseline comparison for evaluating the performances of solutions developed in higher level notations. Such comparison could be used to assess the overhead introduced by mapping tools (and eventually bring to their improvement). On the other hand, we also aim at identifying those features that are missing in PROMELA (e.g., logging and visualization facilities) and that could be added to the language without compromising the performance.

Sect. 2 illustrates how we modelled the case study in PROMELA, and Sect. 3 describes the experiments we conducted. Sect. 4 discusses some problems we faced during the development of the model and some lessons learned. Finally, Sect. 5 reviews some related work and Sect. 6 concludes the paper.

2 Model

2.1 SPIN modelling platform

The PROMELA language is the input language for the SPIN model checker [12]. A PROMELA model consists of global variables and definitions of process types. Each process type can be instantiated resulting in a process (instance), which becomes the active entity of the model. A process consists of local variables and a sequence of statements, which are executed basically in the order in which they are written. The value of local variables and the process program counter define the state of the process. The model is then defined as all allowed inter-

leavings of particular processes' statements; in turn, the (global) model state is the composition of states of all processes and the values of global variables.

SPIN allows for both simulation and verification of the models. While during verification, the entire state space, i.e., all states of the model are explored, the simulation can be seen as one particular execution, i.e., one particular interleaving of processes' statements, similarly to an execution of a multi-threaded program.

SPIN provides several means for specifying, and also verifying specific model properties. They include asserts (as known from imperative programming languages, such as Java and C/C++), LTL formulae, checking for deadlocks and non-progress cycles, and so-called "never claims" [12]. While asserts can be used in a similar way as in common programs to check variable values at particular model places (i.e., expressing safety properties), LTL and non-progress cycles allow the developer for expressing and checking more complex properties, including both safety and liveness ones. Never-claims provide even more precise way for specification of the properties in an imperative way (complementary to declarative LTL).

2.2 Description of model

In this section, we give a general overview of the model we developed.

The model (the `model.pml` file) consists of two main parts, each one represented by a PROMELA process—a `reality` process and a `trackside` process. On the other hand, each train is represented by a data structure (`Train`), thus being a passive entity in our model.

The `reality` process represents the real situation. It manipulates an array of `VSS` (`real[]`) representing the actual position of the trains and updates the fields of the `Train` structure accordingly.

The trains are moved either according to their movement authority in a random manner, or according to a defined scenario. The operation mode and the particular scenario to be executed is determined by the value of the `sce` variable; if none is defined, random mode takes place. The scenarios are specified in a separate file `scenarios.pml`.

The `trackside` process represents the behaviour of the trackside infrastructure. It receives the information reported by trains (by means of reading the fields of the `Train` structure) and changes the state of particular VSSs accordingly. The VSSs and their states are stored in an array (`vss[]`), similarly to the real state (`real[]`).

Code 1 shows an excerpt of the data structures used in the model. Each VSS (i.e., a `VSS`) can be in one of the four states `FREE`, `OCCUPIED`, `AMBIGUOUS`, and `UNKNOWN` as described in the case study document [6]. The VSSs of the `real` array, instead, can be set just either to `FREE` or to `OCCUPIED`; the `real` array is not accessed by the trackside detection system, but it is only used for debugging and verification purposes to compare the assumed state with the real one.

Array `ttd` represents the real state of each TTD section; note that the TTD information is always considered safe by the trackside, and so there is only

```
mtype = {REAL, TRACKSIDE};
mtype schedule = REAL;

mtype = {FREE, OCCUPIED, AMBIGUOUS, UNKNOWN};
typedef VSS {
  mtype state;
  byte ttd;
}

VSS vss[VSSCOUNT];
VSS real[VSSCOUNT];

mtype = {TTDFREE, TTDOCCUPIED};
typedef TTDSection {
  byte firstVSS;
  mtype state;
  bool ghost;
}
```

```
TTDSection ttd[TTDCOUNT];

typedef Train {
  byte front;
  byte rear;
  bool connected;
  bool integer;

  byte eom;
  byte eoma;
  bool hasreported;
  byte reportedposition;
  byte reportedintegrity;
  ...
}

Train trains[TRAINCOUNT];
```

Code 1. Data structures

one copy of the array. In order to model the division of TTDs in VSSs, each `TTDSection` stores the index of the first VSS in the section. Similarly, each `VSS` stores an index of the `TTDSection` of which it is a part.

The `Train` structure involves fields both representing the real situation (e.g., position of the train) and those communicated to the trackside (e.g., reported position). The most important `Train` fields are:

- `front` and `rear` representing the real VSSs containing the front end and the rear end of the train (either the same one, or two consecutive ones)—see Sect. 2.3;
- `eoma` is the end of movement authority that the trackside grants to the train;
- `eom` is the destination of the train (i.e., "end of mission");
- `hasreported` tells whether the train has reported in its last step;
- `reportedposition`, `reportedintegrity`, ... represent the information reported by the train through the PTD; for example, `reportedposition` is the last reported position of the front end of the train.

Often, we use high values (254, 255) to model "unknown" or "invalid" values. For example, if a train has never reported, the last reported position is set 255.

The behaviour of the model is specified by the rules partially shown in Code 2. The model alternatively executes two processes: the `reality` process models the movement of the train, while the `trackside` process models the trackside system. The scheduling is determined by the value of the `schedule` variable. An alternative approach would be using the `d_step` or `atomic` blocks, but this solution brings several issues: (1) when using `d_step` blocks, the non-deterministic choices would not be explored, (2) a statement inside an `atomic` block can become blocked (by mistake in the model), so the other process would get executed (unexpectedly), and (3) when experimenting with `d_step` blocks, spin reported artificial deadlocks. We assume that the last issue is caused by too many statements inside a single `d_step` block.

```
#define rule2A (ttdstate == TTDOCCUPIED) && (trainidonvss != 255) && ...
#define rule7A trainidonvss != 255 && ((mutetimer[0] == 2 && trainidonvss == 0) || ...

...
inline updateVSS(i) {
  ...
  if
    :: vss[i].state == FREE ->
      if
        ...
        :: rule2A -> vss[i].state = OCCUPIED; log("transition #2A taken\n");
        ...
      fi
    :: vss[i].state == OCCUPIED ->
      if
        ...
        :: rule7A ->  vss[i].state = UNKNOWN; log("transition #7A taken\n");
        ...
      fi
    :: vss[i].state == AMBIGUOUS -> ...
    :: vss[i].state == UNKNOWN -> ...
  fi
}

proctype trackside() {
  do
  :: schedule == TRACKSIDE ->
    ...//set timers
    atomic {
      for (i : 0 .. VSSCOUNT - 1) {
        updateVSS(i);
      }
    }
    ...//update end of movement authority
    schedule = REAL;
  :: timeout -> break;
  od;
}

...

proctype reality() {
  do
  ::schedule == REAL ->
    if
      :: (vss[0].state == FREE) && (alive < 2) -> spawntrain(alive); alive++;
      :: trains[0].alive -> move(0); // train 0 moves
      :: trains[1].alive -> move(1); // train 1 moves
      ...
    fi;
    trainreport(0); // train 0 can either report or not
    trainreport(1); // train 1 can either report or not
    schedule = TRACKSIDE;
  od;
}
```

Code 2. Rules

When executed, the `reality` process non-deterministically performs one of the following actions:

- spawn of a new train (up to the fixed maximum train number of 2);
- progress of the spawned trains. Each train can perform one of the following actions:

- if the train occupies only one VSS, it can either move only the front in the next VSS (so occupying two VSSs), or move entirely in the next VSS. The type of movement is chosen non-deterministically;
- if the train spans over two VSSs, the rear of the train can be moved to the VSS containing the front;
- the train disappears if it reaches `eom`, or the end of the modelled part[2];
- the train can decide not to move from the current position. This models the situation in which the train moves while staying inside the same VSS(s);
- the connected train can disconnect and vice versa;
- the train can also split into two trains, if there is just one train in the system so far.

Moreover, at each step, the train can either report or not. The report always includes information about the position of the train, but may or may not involve integrity information.

At each step, the `trackside` process:

1. non-deterministically sets the starting and expiration of timers;
2. updates the states of the VSSs according to the rules of the case study document;
3. updates the `eoma` of the trains up to the first free VSS.

2.3 Abstractions

One of the main difficult aspects in modelling is to decide which details of the requirements can be abstracted away as not necessary for checking the correctness of the system. Leaving out details of the model has two advantages: the model is simpler to understand and maintain, and can be handled by verification tools (i.e., to tackle the state explosion problem). In the following, we report on the abstractions we applied to the case study requirements.

Train length. We do not explicitly model the train length. We assume that a train can fit in a VSS and, therefore, during its journey, it can span at most over two VSSs. This decision is motivated by the specification document [6] (including the scenarios) that only considers these train lengths.

Number of trains. The scenarios also report at most two trains. We assume this situation to be general enough to capture situations with a greater number of trains. As in the case of the train length, we were inspired by the specification document and the motivation to keep the model state space smaller.

[2] Note that the case study assignment [11] considers movement only in one direction, i.e., no backward moves.

Train behaviour. We assume that the train can do at most one action at a time: move the front end, move the rear end, move entirely to the following VSS, disconnect, or reconnect. Therefore, for example, it is not possible that the reality process moves and disconnects a train in a single step. However, we can still model a given combination of train actions in several consecutive steps; for example, the simultaneous train movement and disconnection is captured by two steps, in which the train first moves and then disconnects. Our experiments with scenarios show that this approach includes also all the one-step VSS updates, so we consider it an over-approximation. In addition to that, the train can also disappear after reaching either its `eom` or the end of the modelled railway track.

Timers. The timers are modelled in quite a precise way. Each timer is started and stopped when the conditions for it [6] are met. Since PROMELA does not provide any real time support, the timers in our model non-deterministically expire after they are started. This can lead to unrealistic situations in the model, which would not appear in practice. For example, a train can move over several VSS without reporting its position and without expiring its mute timer. On the other hand, there is no precise relation between the timer expiration time, train speed and VSS lengths in the requirements document [6], which we consider its particular deficiency[3]. Our approach also allows us not to explicitly model the *wait integrity timers*, since they can be covered by the *integrity loss propagation timers*.

3 Experiments

We run all the experiments on a Linux blade server with Xeon X5687 CPU with 192GB RAM. The model file together with the scenario definitions and output of scenario simulations are available at `http://d3s.mff.cuni.cz/~kofron/abz18casestudy.html`. Modelling the whole case study took about one month: two weeks for creating the model and other two weeks for debugging it.

In order to validate our approach, we simulated the nine scenarios reported in the requirements [6]; in order to automatize the approach, we had to specify in the model itself a mechanism for forcing some particular steps: more details are given in Sect. 4. In almost all the steps of all the scenarios, we were able to reproduce the exact VSSs configuration, using the same rules reported in the requirements to update the single VSSs. In some particular steps, instead, our simulation differs because of errors and/or ambiguities in the specification; we detail all of them in Sect. 4.

In addition to validation, we performed a more detailed analysis in terms of formal verification. We ran several verification runs with different settings (stack size limits, storage modes—exhaustive vs. bitstate hashing) to cover as large part of the state space as possible. We learned that the state space is

[3] Formulations in [6] such as "A value between 5-10 seconds would seem to be practical" and "...this timer could be set to a value of at least 27s..." are not of much use.

branching a lot and so that it makes sense to run the verifier with both large and small stack sizes. In sum, we ran the verification over more than a week, being able to explore over 6×10^{11} states. Of course, we are not aware of the total size of the state space, however, several bitstate hashing verifications were successfully accomplished. Even though being just approximative method, no error was found this way.

We attempted at proving a set of safety properties (assertions in the model) regarding the correct movement of trains:

- In order to avoid train collision, we check that a train does not move in a VSS occupied by another train. We actually found a violation of this property when the mute timer of the first train is started and does not expire while the train is moving over several VSSs; in this case, the chasing train can proceed and enter the VSS of the first train[4]. The violation is due to the fact that, as explained in Sect. 2.3, we do not put any constraint on the timer expiration (for example, a timer can start and never expire or it can take arbitrarily long). We think that there should be a relation between the train speed, timers duration, and the lengths of VSSs that, however, is not articulated in the requirements. The assertion violation was found in about 30 seconds, using the stack size of 4,000 states.
- We also check that a train does not move beyond its end of movement authority (EoMA) nor beyond its end of mission (EoM). This property can be violated (and we assume that in practice it is—c.f. step 2 of scenario 8) in the "on-sight mode", which we do not attempt to model. The reason for this is that the safety requirements of the system cannot be guaranteed in this mode.

In addition to functional correctness of the modelled system, we checked whether the requirements are consistent. The conditions specified in the state machine for the VSS (see Sect. 5 in [6]) should guarantee that, for a VSS, it is not possible that two rules bringing to different target states are applicable at the same time; when this is possible, the requirements explicitly specify the priority among the rules. Therefore, the update of VSSs should be deterministic. However, it could still be that the requirements document is not correct or that we wrongly implemented the rules. In order to check that the update of VSSs is deterministic, we performed an additional check. Before updating a VSS (by non-deterministically selecting one rule that is applicable), we count all the rules that are actually applicable and, if more than one applies, we raise an assertion violation. In this way, we found that in a particular scenario two rules are applicable: in step 5 of scenario 8, both rules #10A (the one reported in the requirements document) and #9A can be applied for VSS12. The VSS is the last one of TTD10, it is in state AMBIGUOUS, and a train has just left it and crossed the TTD border. Rule #10A is applicable when "VSS is left by all reporting trains", while #9A when "TTD is free"; both conditions

[4] The simulation output of the assertion violation can be found at `http://d3s.mff.cuni.cz/~kofron/abz18casestudy.html`.

are clearly satisfied in the current situation. The same problem appears in the update of VSS 12 in step 7 of scenario 9. We believe that the problem is due to an ambiguous description of rule #10A in the requirements document; indeed, the description of the rule refers to paragraphs 3.6 and 3.7 that regard non-integer trains; however, in scenarios 8 and 9 both trains are always integer, and, therefore, it is not clear why rule #10A should apply. Also, it is not clear to us what "all reporting trains" refers to—all reporting trains at the same TTD or all reporting trains in the system?

4 Discussion

In this section, we discuss the problems we faced during the model development. In particular, we focus on the problems that are caused by the deficiencies of the adopted modelling language (see Sect. 4.1), and on those that arise when reading the requirements (see Sect. 4.2).

4.1 Missing facilities

PROMELA provides a limited support to debug/log the model by means of standard printing to the standard output. In order to visualize the train movement, we had to add suitable printing outputs into the model. Fig. 1 shows an excerpt of the simulation of a scenario[5]. For each simulation step, we report events related to trains and signals, which VSSs have been updated and by which rule, and EoMAs of existing trains. Moreover, we also visually depict the real position of the trains in the first line (`A`, `a`, `B`, `b` for the first train connected, for the first train disconnected, for the second train connected, and for the second train disconnected, respectively), the VSSs statuses in the second line, and the TDDs statuses in the third line. The last line shows the TTD number.

Although the implemented solution worked pretty well for our purposes, some formal methods provide nicer ways to visualize the model evolution; for example, for the B method, ProB provides an animator [13] that allows to visualize specific pictures associated with model states. The advantages of this method are several: first of all, the visualization can be much nicer and understandable than that obtainable by standard text printing; moreover, since the visualization is defined in a separate function, there is a clear separation of concerns (specification of the behaviour and logging) in the model. As future work, we could consider to add some animation facilities to PROMELA/SPIN.

Another feature that we missed during the development is a proper support for guided simulation. SPIN allows to simulate the model by choosing, at each step, which state to take as next state (by selecting the values of variables that are non-deterministically updated); however, if the model is big, doing a manual

[5] Note that in SPIN, we sometimes need to perform multiple steps in order to model a single step of a scenario reported in the requirements document; therefore, the step numbers (6 and 7) reported in the figure are different from the corresponding steps of the requirements document (steps 3 and 4).

```
Step 6 of scenario 9
Train 1 disconnects
Train 0 reported having left VSS3
Train 0 reported with integrity
Train 1 NOT reported
Timer ghosttimer expired
VSS2: transition #1F taken
VSS3: transition #1F taken
VSS4: transition #8B taken
Train A - eoma: 9
Train B - eoma: 3

b                           A
U       U       U       U       A       F       F       F       F       F
0       0       0       0       0       F       F       F       F       F
0       0       1       1       1       2       2       2       3       3
--------------------------------------------------------------------------------

Step 7 of scenario 9
Moving front of train 0 forward
Moving both front and rear of train 1 forward
Train 0 reported having left VSS3
Train 0 reported with integrity
Train 1 NOT reported
VSS5: transition #3A taken
Train A - eoma: 9
Train B - eoma: 3

        b                   A       A
U       U       U       U       A       A       F       F       F       F
0       0       0       0       0       0       0       0       F       F
0       0       1       1       1       2       2       2       3       3
--------------------------------------------------------------------------------
```

Fig. 1. Two steps (steps 3 and 4) of simulation of Scenario 9

simulation can be particularly cumbersome. Some formal methods allow to specify *scenarios* of the model execution by writing script in which non-deterministic choices are fixed and the model is forced to perform a given number of steps (in a kind of test script). The main advantage of these tools is that scenarios can be executed as many times as necessary (usually, after the model update) in order to check the correctness of the model in that particular situation. In our model, we provided a basic support for scenarios. A scenario is described by means of an array of steps (typedef `Step` shown in Code 3) that specifies the non-deterministic choices to perform at each step. A `Step` is constituted by some variables as `train[]` and `mutetimer[]`; the `train[]` array, for example, encodes which actions should be performed by each train (moving, disconnecting, reporting, etc.). When running the model, we can specify whether it must be run randomly (if variable `sce` is not defined) or if it must read the choices specified in a given scenario `sce`. In order to drive the simulation according to the scenario commands, we had to modify the model such that, in all the points in which a non-deterministic choice is done, the choice specified in the scenario is chosen. Since a scenario variable can encode multiple commands, in the model we use proper masks to extract the commands.

```
typedef Step {
  byte train[2];
  byte mutetimer[2];
  byte disconnecttimer[VSSCOUNT];
  byte integritylosstimer[VSSCOUNT];
  byte shadowtimerA[TTDCOUNT];
  byte shadowtimerB[TTDCOUNT];
  byte ghosttimer;
  byte eoma[2];
  byte eom[2];
}

typedef Scenario {
  Step step[SIMULATIONSTEPS];
}
```

```
inline initScenarios() {
  ...
  //step 3
  scenarios[9].step[6].train[0] = 48;//train 0 reports
  scenarios[9].step[6].ghosttimer = 2;//ghost timer expires
  scenarios[9].step[6].train[1] = 5; //trains 1 disconnects
  //the end of movement authority of train 1 is extended to VSS3
  scenarios[9].step[6].eoma[1] = 3;
  //step 4
  //train 0 moves front and reports
  scenarios[9].step[7].train[0] = (2 | 48);
  //train 1 moves entirely without reporting
  scenarios[9].step[7].train[1] = (8);
  ...
}
```

Code 3. Excerpt of Scenario 9 specification

Such approach allowed us to easily specify all the scenarios reported in the case study document. The main drawback of the approach is that the reading of the scenario had to be hard-coded in the model itself, decreasing the readability and maintainability of the model. As future work, we plan to develop a higher support for scenario (e.g., a DSL for writing scenarios) as, for example, that provided for the ASM method [5].

4.2 Issues in modelling the requirements

One of the advantages of adopting formal methods is that they allow to highlight the inconsistencies and/or ambiguities contained in the requirements. Although the case study document [6] is already quite detailed, there are still some parts that we had problem in understanding. In the following, we review all these issues and describe how we handled them.

Delay in TTD processing. According to the requirements [6], the "TTD information is considered as safe", i.e., it reports "free only if no train is present on the TTD section". Therefore, on the base of this requirement, we always consider the TTD information trustworthy; however, step 7 of scenario 5 reports a case in which "due to the delay time of the TTD detection system, the TTD is still considered occupied". We agree that the information provided in the requirements is not stating that the TTD is occupied only if the train is present; however, we think that the requirement is ambiguous and can bring to this misunderstanding. We are not sure about the length of the delay with respect to the train speed, report frequency, etc., to understand if it is important to consider this particular delay in the model. Therefore, since we are still not sure about which should be the correct behaviour, we decided to keep the model that we produced starting from the reading of the requirements and so we free the TTD section as soon as the train leaves it; therefore, we do not support step 7 of scenario 5.

Updating the end of movement authority. The requirements [6] do not exactly specify how and when the *end of movement authority* (EoMA) is modified and

by whom. In the model, we assumed that the EoMA is modified by the track side authority after each update of the VSSs states: the EoMA is extended as long as the VSSs are free or unknown or end of movement EoM is not reached. This is also motivated by the specification scenarios and it seems to be the most permissive choice that still preserves the safety of the system.

Order of update. At first, we assumed that the update of VSSs depends on the previous state of the other VSSs; actually, this seems to be not true. In step 3 of scenario 9, VSS23 must become ambiguous if the previous VSS is unknown; however, the previous VSS becomes unknown in the same step. Therefore, we update the VSSs from left to right; however, we are not sure whether this assumption is correct.

Loosing the integrity after train split. The requirements do not specify which is the integrity status of a train (actually the integrity status of the two parts of the train) when it splits. At first, we assumed that the train can be either integer (if it splits on purpose and it is aware of its integrity) or non-integer (if it splits accidentally). However, from scenario 5, it seems that the train always looses its integrity when it splits, and so we modelled this behaviour.

On sight mode. Requirements [6] mention the possibility for a train to operate in *on sight* (OS) mode "that gives the driver partial responsibility for the safe control of his train" [9]. We do not support the OS mode in the model, because handling it would not allow any kind of safety check regarding the correct operation of trackside detection system (as the driver could bring the train in an unsafe situation). However, we support the OS mode in scenarios, in which we can force the train to perform a given not allowed movement, as proceeding after its eoma; in this way, we have been able to reproduce step 2 of scenario 8.

Inconsistencies in the scenarios. We identified some inconsistencies in some scenarios that report wrong rules for the reported state transitions of the VSS. In particular, in steps 6 and 7 of scenario 8, rule #2A is used to modify VSS22 and VSS23 from UNKNOWN to OCCUPIED; however, rule #2A goes from FREE to OCCUPIED. In other cases, instead, we think that the scenarios are not precise and they do not mention an additional transition that must be taken before the reported one:

- in step 8 of scenario 6, rule #6A is used to modify VSS23 from AMBIGUOUS to FREE (however, rule #6A goes from OCCUPIED to FREE); we think that the requirements imply that rule #11 must be taken before.
- in step 5 of scenario 8, rule #11A is used to modify VSS21 from UNKNOWN to OCCUPIED (however, rule #11A goes from AMBIGUOUS to OCCUPIED); we think that the requirements imply that rule #5 must be taken before. The same issue appears in step 7 of scenario 9 for VSS21.

5 Related work

At the time of writing, we are not aware of any formalization and/or validation and verification of the Hybrid ERTMS/ETCS Level 3 system.

Regarding the SPIN model checker, it has been applied to the modelling and verification of different safety-critical systems[6]; Havelund et al. [10], for example, applied it to the verification of the multithreaded plan execution module of an artificial intelligence-based spacecraft control system architecture part of the DEEP SPACE 1 mission.

A common approach in model checking models developed in high-level notations is to exploit existing model checkers as SPIN, NuSMV, UPPAAL, etc. For example, in [3], the authors discuss the advantages of using high-level notations in hardware design. They observe that HW designers are used to high level notations as Bluespec and they have problems when dealing with the lower level notations; the authors claim that the notation of the verification tool should be transparent to the designer, who should specify the model and the properties in the same high level notation, without caring about the intricacies of the notation of the verification language. With this aim, a common approach is to automatically translate high level models in models of existing model checkers; this requires to define the mapping from the source notation to the target notation, and also a reverse translation of the counterexamples returned by the model checkers in concepts of the source notation.

However, such translation often introduces an overhead that affects the scalability of the verification; for example, this is reported for translation of ASMs to NuSMV [2], of UML models to PROMELA [7], and of Simulink models to NuSMV [16].

On the other hand, a more recent approach is to develop model checkers directly handling the high level notation, as ProB that directly model checks B models. In [14], the model checker ProB is compared with SPIN. The author notices that, whenever the number of states of a B model and a PROMELA model are the same (models developed for the same problem), SPIN outperforms ProB of several orders of magnitude, as SPIN performs verification directly in C and it employs several optimizations (e.g., partial order reduction, bitstate hashing), while ProB uses an interpreter written in Prolog. On the other hand, the author also shows that, in some cases, the ProB model checker behaves better as it avoids the state explosion occurring in SPIN (if the `atomic` construct is not used), it employs a mixed depth-first breadth-first strategy, and it exploits symmetries present in high-level models.

6 Conclusions

In this paper, we proposed the modelling, validation, and verification of the Hybrid ERTMS/ETCS Level 3 Case Study in SPIN. The tool allowed us to model

[6] `http://spinroot.com/spin/success.html`

all the requirements of the case study, reproduce all the scenarios reported in the case study document, and verify the model. We have shown that, although very powerful in terms of verification, SPIN misses some facilities (logging and scenario specification) that could help in debugging and validating the model. For this work, we devised an approach to encode scenarios that, however, requires to modify the model itself; as future work, we plan to design a language for writing scenarios (as test cases) and implement a tool that, given a model and a scenario for it, drives the SPIN simulation over the model as specified in the scenario.

References

1. J. Abrial. *Modeling in Event-B – System and Software Engineering*. Cambridge University Press, 2010.
2. P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In *Proceedings of the 2nd International Conference on Abstract State Machines, Alloy, B and Z (ABZ 2010)*, volume 5977 of *LNCS*, pages 61–74. Springer, 2010.
3. Arvind, N. Dave, and M. Katelman. Getting formal verification into design flow. In J. Cuellar, T. Maibaum, and K. Sere, editors, *FM 2008: Formal Methods: 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008 Proceedings*, pages 12–32, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
4. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
5. A. Carioni, A. Gargantini, E. Riccobene, and P. Scandurra. A scenario-based validation language for asms. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *Abstract State Machines, B and Z: First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, pages 71–84, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
6. Hybrid ERTMS/ETCS Level 3. Technical report, EEIG ERTMS Users Group, 07 2017.
7. J. Chen and H. Cui. Translation from adapted UML to Promela for CORBA-based applications. In S. Graf and L. Mounier, editors, *Model Checking Software*, pages 234–251, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
8. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*. Springer, July 2002.
9. Glossary of terms and abbreviations. Technical report, ERA * UNISIG * EEIG ERTMS USERS GROUP, 5 2016.
10. K. Havelund, M. Lowry, and J. Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineering*, 27(8):749–765, Aug 2001.
11. T. S. Hoang, M. Butler, and K. Reichl. The hybrid ertms/etcs level 3 case study. Technical report, 2018.
12. G. J. Holzmann. *The SPIN Model Checker – Primer and Reference Manual*. Addison-Wesley, 2004.
13. L. Ladenberger, J. Bendisposto, and M. Leuschel. Visualising Event-B models with B-Motion Studio. In M. Alpuente, B. Cook, and C. Joubert, editors, *Formal Methods for Industrial Critical Systems: 14th International Workshop, FMICS*

2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings, pages 202–204, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
14. M. Leuschel. The high road to formal validation. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *Abstract State Machines, B and Z: First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, pages 4–23, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
15. M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, Mar 2008.
16. B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for translating Simulink models into input language of a model checker. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering*, pages 606–620, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
17. A. Prigent, F. Cassez, P. Dhaussy, and O. Roux. Extending the translation from SDL to Promela. In D. Bošnački and S. Leue, editors, *Model Checking Software*, pages 79–94, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.