Noname manuscript No.
(will be inserted by the editor)

Validating the Hybrid ERTMS/ETCS Level 3 Concept with Electrum

Alcino Cunha · Nuno Macedo

Received: date / Accepted: date

Abstract This paper reports on the development of a formal model for the Hybrid ERTMS/ETCS Level 3 concept in Electrum, a lightweight formal specification language that extends Alloy with mutable relations and temporal logic operators. We show how Electrum and its Analyzer can be used to perform scenario exploration to validate this model, namely to check that all the operational scenarios described in the reference document are admissible, and to reason about expected safety properties, which can be easily specified and model checked for arbitrary track configurations. We also show how the Analyzer can be used to depict scenarios (and counter-examples) in a graphical notation that is logic-agnostic, making them understandable by stakeholders without expertise in formal specification.

Keywords Formal specification · Validation & Verification

1 Introduction

This paper reports on the modelling and subsequent validation and verification of the Hybrid ERTMS/ETCS Level 3 (HL3) concept [10] in Electrum [19], developed as an answer to the ABZ'18 call for case study contributions. Electrum is a lightweight formal specification language that extends Alloy [17] with mutable relations and linear temporal logic (LTL) operators. The result is a language as simple and flexible as Alloy, but with improved support for the specification of reactive systems and for the model checking of safety and liveness properties specified in LTL (possibly using past operators as well). Its Analyzer [16] provides support for bounded (through SAT) and complete (through SMV) model checking, with scenarios (and counter-examples) presented back to the user in a unified graphical interface, customizable by user-defined themes.

The main features and outcomes of this work are:

- the development of a discrete HL3 model for arbitrary track configurations encoding most relevant features, including the VSS state machine, TTD and PTD communication delays, non-integer trains, timers and movement authorities
- this model is amenable to validation and verification with automatic analyses, including support for animation and scenario exploration
- the development of a dedicated visualization theme that allows instances to be inspected in a logic-agnostic manner that resembles the one used in the reference document to present scenarios [10]
- the encoding of all the reference scenarios [10], which further validate the model and show that the VSS state machine (mostly) acts as intended
- the specification of simple safety properties for the HL3 and their subsequent automatic verification under multiple configurations, both with bounded and complete model checking

Model development was carried by the two authors of the paper, both with extensive background in Alloy and proponents of the Electrum extension. The challenging nature of the work allowed the team to identify and implement improvements to Electrum and its Analyzer. It should be noted that the authors had no *a priori* domain knowledge, and that the work was mainly based on the provided reference document for the HL3 [10].

This paper is an extended version of a conference paper [7]. Since then, the HL3 reference document has been updated from version 1A [9] to 1C [10], which

A. Cunha · N. Macedo
INESC TEC & Universidade do Minho, Portugal

changed certain components of the system and clarified certain nomenclature ambiguities. We updated our model accordingly, and also took the opportunity to implement an additional feature (namely, TTD communication delays) and address certain methodology issues (namely, the traceability between the reference document and the model). Unless explicitly mentioned, the model referred to in this paper is the 1C version.

Section 2 exposes the modelling strategy employed. The resulting HL3 model, as well as relevant design decisions, are presented in Section 3. Electrum concepts are presented throughout the section as needed. Section 4 describes how the model was validated using the Analyzer, including the encoding of the operational scenarios, and explores some desirable safety properties that were automatically verified. Section 5 discusses issues identified in the HL3 and evaluates the employed methodology. Section 6 compares our work with other answers to the challenge and other relevant work, while Section 7 points directions to future work.

2 Requirements and Modelling Strategy

Our modelling strategy was mostly based on the HL3 concept reference document (initially version 1A [9], currently adapted to version 1C [10]), and occasionally on the ERTMS/ETCS glossary [11] to clarify concepts. The current model also incorporates some environment assumptions from the introductory document for the ABZ'18 case study challenge [14] that are not explicit in the reference document.

The main focus of the HL3 concept (and consequently, the main target of our analysis) is the design of the VSS management sub-system of the trackside system, with the remainder sub-system (handling MA authorisations) and the interlocking and train states (with the respective TTD and PTD reports) acting as the environment (see Fig. 1).

The VSS management sub-system requirements are mostly encoded directly in our model, including how the TTD and PTD reports are interpreted [10, §3.3,§3.5], the VSS state machine [10, §3.2,§5] and the definition of timer events [10, §3.4]. This modelling process was backed by the remainder sections [10, §3.6–§3.11,§4,§6] that provide justifications and examples for the state machine and timer events.

The MA authorisation sub-system is outside the scope of the HL3 reference document and its behaviour depends on particular implementation decisions. To model it we tried to infer sensible assumptions from the justification of the VSS sub-system design decisions and its behaviour in the operational scenarios. Some assumptions that were more clearly formalized in [14], unavailable at the time of the original publication, were also integrated. The same strategy was followed for the remainder environment assumptions, i.e., the interlocking and train behaviour. Nonetheless, the environment is purposely left under-specified, to allow a wide range of alternative behaviours.

Due to the complexity (and occasional ambiguity) of the requirements, the modelling process was iterative with the encoding of the operational scenarios, which allowed for the validation of the formalization. Inconsistencies between the reference behaviour of scenarios and that of our model usually evidenced issues with the formalization due to our limited knowledge of the domain or ambiguities in the reference document. Occasionally, we found no reasonable interpretation of the reference document, and assumed an inconsistency in the requirements. All such issues found in version 1A have been fixed in version 1C of the HL3 reference document; they are identified in Section 5 for completeness. Others were identified in version 1C, which are also discussed in Section 5.

Once the model was stable, it was amenable to the verification of desirable properties. Neither reference document clearly specifies desirable safety properties for the VSS sub-system, so we focused on simple proof-of-concept correctness properties concerning the expected assignment of states to the VSSs. We then moved to the more general property of avoiding train collisions. Due to the open nature of the environment model, this stage required exploring sensible restrictions to it (e.g., how the trains behave in relation to the assigned MAs).

Accordingly, our model is (informally) structured in the following blocks, as depicted in Fig. 1:

- the modelling of the environment assumptions, including the MA management sub-system, and train and trackside state and reporting, loosely specified to allow various alternative behaviours
- the modelling of the VSS management sub-system requirements, including VSS states and timers and their evolution (VSS state machine and timer events)
- the codification of the operational scenarios and animation commands for validation
- the specification of safety properties and checking commands for verification

Traceability between our 1A model and the requirements was mostly provided through nomenclature conventions, clearly identifying each transition of the VSS state machine and timer event with a matching predicate. However, when updating the model to version 1C, which clarifies how the train state and location is interpreted according to PTD reports, the lack of pointers to

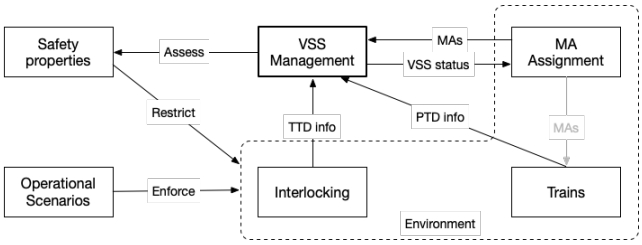

Fig. 1 Architecture of the HL3 model (adapted from [14]).

the reference document proved troublesome. To tackle this issue, we also took the opportunity to annotate the elements of the 1C model with the reference document sections where the requirements were identified.

Proper abstraction is key to achieve a model that is representative of the system under study but that is still prone to being automatically analysed for relevant properties and easily understood by all interested parties. In our model, the main abstraction points arise from the mismatch between certain continuous aspects of the rail traffic management domain and the necessarily discrete nature of state-based modelling languages like Electrum. These include concerns with train length changes, as well as real-time issues related to communication delays and the use of timers to optimize the performance of the system. Nonetheless, since the level of granularity of HL3 is that of the VSS, this abstraction has not prevented us from successfully formalising the VSS sub-system and supporting every operational scenario.

When processing PTD reports, version 1C of the reference document specifies 10 steps through which the state machine is updated, each processing a different kind of information. Forcing such procedure at all times would lead to an explosion of the trace lengths of the instances, possibly rendering the problem intractable. Thus, while also allowing such phased evaluation of reports, in the general case our approach assumes all steps to be processed in a single transition (sec Section 3.2 for details). In fact, steps are also often collapsed in a single transition in the description of the scenarios in the reference document, since this usually does not affect the outcome. When it does, it is explicit in the definition of the scenario, and in such cases (Scenarios 7, 8 and 9) our encoding forced the update in individual steps.

Our model does not explicitly consider the classes of trains that are to be supported by the HL3 according to [14] (ERTMS with TIMS, ERTMS without TIMS, and non-ERTMS). In fact, these categories are never explicitly mentioned in the HL3 reference document. Our model considers trains that fail to communicate or report integrity, thus particular classes can be simulated by enforcing additional restrictions: non-TIMS trains must that never report integrity, and non-ERTMS must never communicate with the trackside.

```
open util/ordering[TTD] as D                          // ASM1/2
open util/ordering[VSS] as V                           // ASM3

sig VSS {}
sig TTD { start,end : one VSS }                        // ASM3

fact trackSections {                                   // ASM1,2
  all d:TTD | (d.end).gte[d.start]
  all d:TTD-D/last | d.end.V/next = (d.D/next).start
  D/first.start = V/first and D/last.end = V/last }

fun VSSs[t:TTD] : set VSS {                            // ASM3
  t.start.*V/next&t.end.*(~V/next) }
fun parent[v:VSS] : one TTD {                          // ASM3
  max[(v.*V/prev).~start] }
```

Fig. 2 Excerpt of the static environment.

3 Model Details

The section presents the HL3 Electrum model developed for the HL3 concept 1C, which is available online.[1] Relevant design decisions are explained as the model is presented. The Electrum language is also presented by example throughout the section. The formal presentation of its syntax and semantics are presented elsewhere [19].

3.1 Environment Modelling

In Electrum, likewise Alloy, structure is introduced through the declaration of *signatures* (keyword **sig**), that represent sets of uninterpreted atoms, and *fields*, that create relationships between multiple atoms. Both signatures and fields can be restricted by simple multiplicity constraints (e.g., **set**, **some** or **one**), and a signature hierarchy can be introduced by inclusion (**in**) or extension (**extends**), the latter forcing children signatures to be disjoint. In Electrum signatures and fields may either be static (by default) or variable (those marked as **var**). Static elements represent the possible configurations on which a system can act. During the analysis process, they will be populated with arbitrary atoms for which the structural constraints hold, within a given scope, and then remain frozen throughout the instance traces.

In the HL3 model such configurations are part of the environment specification – depicted in Fig. 2 – and represent the valid partitioning of tracks into train detection sections (signature `TTD`) and virtual subsections (signature `VSS`). HL3 addresses single straight lines, whose granularity is that of the VSS, thus tracks are simply discrete sequences of VSS atoms [14, ASM1–3].

[1] `http://haslab.github.io/Electrum/ertms_1C.ele`

In Electrum this can be achieved by imposing total orders through the library module `util/ordering`, which introduces relations `first`, `last`, `next` and `prev`, qualified for TTDs (`D`) and VSSs (`V`). To partition TTDs, fields `start` and `end` each register exactly **`one`** `VSS` in which each `TTD` starts and ends. This environment representation does not consider any particular dimension of blocks or trains, which mostly does not affect the concept; the few exceptions are identified when relevant.

Relational expressions combine signatures and fields (and constants, namely the empty (**`none`**) and universe (**`univ`**) sets, and the identity binary relation (**`iden`**)) using standard relational operators like union (`+`), intersection (`&`), difference (`-`), join (`.`) or the binary converse ($\sim$), and transitive (`^`) or reflexive-transitive ($*$) closure operators. Relational expressions can also be constructed by *comprehension*. Primitive relational formulas are either inclusion (**`in`**) or equality (`=`) tests, or basic multiplicity tests (e.g., **`some`** or **`no`**), which can be combined through common Boolean operators (**`not`**, **`and`**, **`or`**, **`implies`** or **`iff`**) and first-order quantifications (e.g., **`all`** or **`some`**).

Through **`fact`** paragraphs arbitrary relational formulas can be imposed as axioms that always hold in a model. Such is the case of `trackSections` that guarantees that TTDs are correctly partitioned into VSSs, namely by quantifying over **`all`** `TTD` elements and forcing their end to occur after their start (using the total order on `VSS`), that the last VSS of a TTD is succeeded by the first VSS of the next TTD (except in the edges of the track), and that the edges of the `TDD` and `VSS` orders coincide. This declarative definition eases the analysis of properties over every valid track partition within a given scope for signatures `TTD` and `VSS`. *Functions* (**`fun`**) and *predicates* (**`pred`**) declare reusable expressions and formulas, respectively. For instance, function `VSSs` calculates all the subsections of a `TTD` using transitive closure over the total order on `VSS`, and `parent` retrieves the `TTD` to which a `VSS` belongs by calculating the maximum TTD among those starting before it.

The static structure, presented so far, is essentially a normal Alloy model. Electrum enables the encoding of dynamic structure through variable signatures and fields, declared using the keyword **`var`**. In the HL3 environment these regard the interlocking state, the physical state of the trains and on-board systems, the TTD and PTD information reported at each time, and the currently assigned MAs. An excerpt is depicted in Fig. 3.

The HL3 concept assumes TTD occupation reporting to be safe [10, §3.1.1.5] but with possible delays [10, §4.5.1.3.5]. Although in general a mutable boolean field could be used to encode whether TTDs are reporting, in Electrum this can be achieved by variable sub-signatures.

```
var sig Reports in TTD {}                          // §4.5.1.3.5

abstract sig Train {
  var pos_frnt,pos_rear : one VSS,                   // §3.3.1.1
  var EoA               : lone VSS }           // §3.1.1.7,ASM16

var sig MissStartd,MissEnded,                        // §4.2.1.1
        UnknwnTrain extends Train {}
var sig Reporting extends MissStartd {}              // §1.2.1.1
var sig IntgrtyConfirmed,                   // §1.2.1.1,§1.2.3.1
        IntgrtyLost extends Reporting {}

fact TTDReports { always                       // sensible delays
  all t:TTD | t not in Reports implies t in Reports' }

fun pos : Train → VSS { pos_rear + pos_frnt }

fun occupied : set TTD {                       // §3.1.1.5,ASM4/5
  { t:TTD | t in Reports
    implies some VSSs[t]&Train.pos
    else previously some VSSs[t]&Train.pos } }

fact trainEvolution { always          // inferred from scenarios
  all t:Train | som[t] or eom[t] or move[t]
                or some s:Train | split[t,s] or split[s,t] }

pred move [t:Train] {                 // inferred from scenarios
  t.pos_frnt' in t.pos_frnt.(iden+next)
  t.pos_rear' in t.pos_frnt'.(iden+prev)
  t.pos_rear' in t.pos_rear.(iden+next)
  t in MissStartd iff t in MissStartd'
  t in MissEnded iff t in MissEnded' }

pred twocarr[t1,t2:Train] { historically {
  t1.EoA = t2.EoA and t1.pos_frnt = t2.pos_frnt and ... } }
pred split [t,t1:Train] {             // inferred from scenarios
  twocarr[t1,t2]
  t1 in MissStartd implies t1 in IntgrtyLost'
  t2 in UnknwnTrain' - UnknwnTrain
  t1 in MissStartd iff t1 in MissStartd'
  t1 in MissEnded iff t1 in MissEnded'
  t1.pos_frnt' = t1.pos_frnt and t1.pos_rear' = t2.pos_rear
  t2.pos_frnt' = t2.pos_frnt and t2.pos_rear' = t2.pos_rear }

fun MA[t:Train] : set VSS {                          // §3.1.1.7
  memorisedRear[t].*V/next&(t.EoA).*V/prev }

pred OS[t:Train] { t.EoA = last }                 // §3.10.1.1

fact MAAssigment { let Off = Disconnected+UnknwnTrain | always {
  all disj t1,t2:Train |                                 // ASM17
    not (twocarr[t1,t2] or OS[t1] or OS[t2]) implies
      no MA[t1]&MA[t2]
  all t:MissStartd-Off |                               // §3.2.1.4
    (t.EoA != t.EoA' and not after OS[t]) implies
      (locatedFrnt[t].^next&t.EoA'.*prev).state in Free
  all t:Train | no t.EoA implies                   // §3.1.1.7.1
    (t in Off or previously no t.EoA) } }
```

Fig. 3 Excerpt of the dynamic environment model.

Concretely, signature `Reports` is a subset of `TTD` denoting at each instant which TTDs are reporting.

Signature `Train` denotes the available trains and their state. Each train has an exact physical position (not necessarily known by the trackside) for its front and rear ends [10, §3.3], represented by variable fields `pos_frnt` and `pos_rear`, that point to exactly **`one`** `VSS` at each time. For simplicity purposes, all trains are assumed to be in the track at all times, so trains may not enter or leave the track. Modelling such behaviour can be easily done by creating additional "dummy" VSSs at the beginning or end of the track, as in Scenarios 8 and 9. Variable

sub-signatures are used to represent the state of the PTD communication between each train and the trackside. Trains have either started (`MissStartd`) or ended (`MissEnded`) a mission [10, §4.2.1], or are completely unknown to the trackside (`UnknwnTrain`), for example, when a carriage splits. By being defined by extension (rather than inclusion), each train belongs only to one of these signatures in each instant; by defining `Train` as abstract, each train must necessarily belong to one of them. Trains on a mission may in turn be `Reporting` PTD information at each instant, and additionally report `IntgrtyConfirmed` or `IntgrtyLost` [10, §1.2].[2] Trains not reporting in an instant represent PTD communication delays; trains reporting but without integrity information represent trains that have been unable to confirm integrity. Finally, each train is assigned at most one (**`lone`**) VSS as its current end of authority (EoA), which determines its MA [10, §3.1.1.7].

Relational expressions in Electrum are extended to support primed expressions, that denote their value in the succeeding state, while formulas are extended with future (e.g., **`after`**, **`eventually`** or **`always`**) and past (e.g., **`previously`**, **`once`** or **`historically`**) LTL operators.

Using facts, Electrum supports the imposition of arbitrary temporal restrictions, like fairness conditions, over variable elements. For example, to keep our model manageable, fact `TTDReports` forces TTDs to **`always`** have communication delays of at most one step by stating that a TTD not in `Reports`, must be in it in the succeeding state. Function `pos` abbreviates the union of the front and rear end positions of a train. Auxiliary functions and predicates can also use temporal operators freely. For instance, at each instant, function `occupied` calculates by comprehension all TTDs whose state is occupied: if a TTD reports, then it is occupied if there is any train in it; otherwise, compute its occupancy in the previous state using the past operator **`previously`**.

Restrictions must also be imposed to constraint the behaviour of trains. By analysing the reference document and the operational scenarios, we identified 4 events that affect the state of trains. In Electrum events can be encoded as declarative predicates that relate the current state with the succeeding one through primed expressions (although more advanced events may use full LTL). Start of mission (`som`) and end of mission (`eom`) actions simply update the mission status of a train accordingly. A `move` action updates the physical position of the train. Lastly, a `split` action models the breaking up of a trains composed by two carriages. Fact `trainEvolution` then forces the state of every train to change at each state through one of these events. The PTD reporting signatures (`Reporting`, `IntgrtyConfirmed` and `IntgrtyLost`) are left unrestricted by the events, meaning that there are no restrictions on how often communication or integrity problems occur.

Predicates `move` and `split` are presented in more detail in Fig. 3 (the others are omitted). To keep the evolution of the system manageable, each train is allowed to move forward at most one subsection in each step, and the rear is always kept at most one subsection away from the front. These restrictions could easily be relaxed, but note that since trains may fail to report PTD information, jumping behaviour in the perspective of the trackside still occurs (Scenario 8). The last formulas preserve the mission status of the train when moving. Even though the train behaviour should consider the assigned MA, according to the reference document trains cannot be assumed to stay within the assigned MA [10, §1.2.3.3], so `move` enforces no such restriction. To model the `split` action, two-carriage trains must be identified in the model. In order to avoid introducing additional variables in the model, two trains are assumed to be connected if they had exactly the same state up to that point. This is encoded by predicate `twocarr` that tests the state of two trains using **`historically`**. During break up, the front one will fail to confirm integrity and the rear one will become unknown to the trackside. Other than that, the state of the trains does not change.

Lastly, the MA assignments are modelled. The VSSs covered by an MA are calculated by function `MA` using the transitive closure between the `EoA` and the train location. Trains can also be assigned on-sight (OS) MAs, in which case they are allowed to move freely in the complete track. For simplification purposes, a train is assumed to have an OS MA when its `EoA` is the last VSS of the track, as encoded in predicate `OS`. Finally, loose, but sensible, MA policies based on the reference documents are enforced, even though they are beyond the HL3 concept. MAs are allowed to change as long as the (declarative) constraints imposed by fact `MAAssigment` hold: i) MAs must not overlap at each instant [14, ASM17] (unless they represent a two-carriage train or have OS MAs), ii) an MA can be changed only to a free section of the track [10, §3.2.1.4] (unless an OS MA is assigned), and iii) an MA may only be removed for disconnected trains [10, §3.1.1.7.1]. Note that the MA sub-system consumes information (like train location) from the VSS sub-system that will be presented in the next section.

3.2 VSS Management Sub-system

The VSS sub-system interprets TTD and PTD reports in order to approximate the current location of trains

[2] A third integrity issue in HL3 arises train length changes are reported; since they are treated exactly as integrity lost reports, for simplicity we encode only the latter in our model.

```
one sig VSSMgr {
  var mem_fr,mem_re : Train → one VSS,                  // §3.3
  var jumpng        : VSS → lone Train,              // §3.3.3.6
  var state         : VSS → one State }              // §3.2.1.1

enum State { Unknown,Free,Ambiguous,Occupied }

var sig MuteExpired in MissStartd {}                   // §3.4.1
...
var sig DiscPropRunning,DiscPropExpired in VSS {}      // §3.4.2
...
fun Disconnected : set Train {                        // §3.3.1.3
  MissEnded + MuteExpired }

fun NonInteger : set Train {                          // §3.5.1.3
  IntgrtyLost + WaitIntgrtyExpired }

fun locatedFrnt[t:Train] : one VSS {                   // §3.3.2
  t in Disconnected+UnknwnTrain implies none
  else t.(VSSMgr.mem_fr) }
fun locatedRear[t:Train] : one VSS { ... }             // §3.3.3
fun located[t:Train] : set VSS {                      // §3.3.2.1
  locatedRear[t].*V/next & locatedFrnt[t].*~V/next }

fun assumedRear[t:Train] : one VSS {                   // §3.3.4
  t in NonInteger => locatedFrnt[t] else locatedRear[t] }

fun memorisedFrnt[t:Train] : one VSS {                 // §3.3.2
  t.(VSSMgr.mem_fr) }
fun memorisedRear[t:Train] : one VSS { ... }           // §3.3.3
fun memorised[t:Train] : set VSS { ... }              // §3.3.1.3

fact jumpingTrains {                                  // §3.3.3.6
  always VSSMgr.jumpng' = { v:TTD.start,t:Train {
    t not in IntgrtyConfirmed'
    t in located[parent[v].prev]
    parent[v].prev in occupied - occupied'
    parent[v] in occupied' } } }

fact memoryUpdate {                                    // §3.3
  let fr = VSSMgr.mem_fr,re = VSSMgr.mem_re,jp = VSSMgr.jumpng |
    always all t:Train {
      t.fr' = t in Reporting' implies t.pos_frnt'
              else t not in Disconnected' implies max[jp'.t+t.fr]
              else t.fr
      t.re' = t in IntgrtyConfirmed' implies t.pos_rear'
              else t not in Disconnected' implies max[jp'.t+t.re]
              else t.re } }
```

Fig. 4 Excerpt of location processing.

and trigger timer events, finally updating the state of the VSSs accordingly. An excerpt of its specification is presented in Figs. 4 (location processing) and 5 (state machine and timer events).

The state of this sub-system is registered in the (singleton) `VSSMgr` signature. Fields `mem_fr`/`mem_re` store the memorised front/rear locations of each train. HL3 detects trains "jumping" between TTDs to avoid losing their location due to communication delays [10, §3.3.3.6]; when such an event is detected for a train, it is registered in field `jumpng`. Field `state` stores the state assigned to each VSS [10, §3.2.1.1], which will be calculated from the processed location information. The 4 possible states are represented by the enumeration `State` (a signature partitioned into singleton sub-signatures).

To avoid performance deterioration due to communication fluctuations, HL3 implements a set of timers to avoid unnecessary VSS state transitions. Each of these timers has start and (possibly) stop events, and is assigned to either a VSS, a TTD or a train. All 7 types of timers were implemented in our model. A set of variable signatures identifies which timers with non-trivial start/stop events are running in each instant (e.g., `DiscPropRunning` contains all VSSs whose disconnect propagation timer is running). It is easy to identify mute and integrity lost timers by inspecting the current state, so their running signatures were not created. A second set of signatures then registers timers that have actually expired (e.g., `MuteExpired` for all trains whose mute timers expired, and `DiscPropExpired` for VSSs with expired disconnect propagation timers).

The HL3 concept treats trains differently depending on the available PTD information. Auxiliary functions determine when a train should be treated as `Disconnected` – mission ended or mute timer expired [10, §3.3.1.3] – or as `NonInteger` – integrity loss reported or wait integrity timer expired [10, §3.5.1.3]. Fact `jumpingTrains` determines, by comprehension, the state of field `jumpng` at each instant: a train was located in a TTD that became free, and the succeeding is still free as well. The way the train locations are approximated depends on the inferred status and detected jumps, and is encoded in fact `memoryUpdate` [10, §3.3]: if the train is reporting, the front end location is updated to the current physical location of the train (and also the rear end location, if reporting integrity); otherwise, if the train is not disconnected, try to identify TTD jumps; otherwise keep the memorised location unchanged.

Auxiliary functions to align the processing of location information in our model with the nomenclature from the reference document. Although the memorised location of a train is stored, it is in general not used by the VSS sub-system when a train is disconnected [10, §3.3.1.3]. This is encoded in functions `locationFrnt` and `locationRear`, for the front and rear end of a train, respectively; function `located` then calculates all VSSs occupied by a train between its rear and front ends. In the cases where the memorised location is to be considered, the field can be accessed directly (corresponding functions `memorisedFrnt`, `memorisedEnd` and `memorised`). Finally, when integrity information is missing, there is a notion of assumed rear location (used in a single transition, #10A) that approximates the rear end from the known train length [10, §3.3.4]. Since our model abstracts length-related information, initially we took a conservative approach where the assumed rear was the VSS before the front end location (recall that our model assumes, as do the operational scenarios, a train occupies at most two VSSs). However, this assumption broke operational scenarios (e.g., Scenario 9), where the assumed rear happens to be the same VSS as the front

```
fact setTimers { always {                                    // §3.4
  setMuteTimer and setDiscPropTimer and ... } }

pred setMuteTimer {                                  // §3.4.1.2
  MuteExpired in MissStartd-Reporting }
...
pred setDiscPropTimer {                              // §3.4.2.2
  DiscPropExpired in DiscPropRunning
  DiscPropRunning' =
    (DiscPropRunning+DiscPropStart-DiscPropStop)-DiscPropExpired
  no DiscPropExpired & DiscPropExpired' }
fun DiscPropStart : set VSS {
  { v:VSS | some t : Train {
    (v in MA[t] and v.state' = Unknown and
                    t in MuteExpired'-MuteExpired) or
    (v in located[t] and eom[t]) or
    (v in located[t] and t in MuteExpired'-MuteExpired) } } }
fun DiscPropStop : set VSS { ... }
...
pred n04 [v:VSS] {                                    // §5.1.1.1
  v.state = Unknown
  n04A[v] or n04B[v] }
pred n04A [v:VSS] {                             // §5.1.1.1, #4A
  parent[v] not in occupied' }
pred n04B [v:VSS] {                             // §5.1.1.1, #4B
  some tr:(MA[v])' {
    tr in Disconnected-Disconnected'
    v in (locatedFrnt[tr])'.^next } }
...
fact stateMachine {                                   // §5.1.1.1
  always all v:VSS | v.state' = (n01[v] implies Unknown else
                                 n02[v] implies Occupied else
                                 n03[v] implies Ambiguous else
                                 n04[v] implies Free else
                                 n12[v] implies Occupied else
                                 ...
                                 n11[v] implies Occupied else
                                                v.state) }
```

Fig. 5 Excerpt of timer events and the state machine.

end location. Thus our current encoding uses the front end location as the assumed rear end location, although this is a potential treat; a possible improvement would be to introduce (abstract) train lengths that could be set to concrete values when needed.

For each of the 7 timer types there is a predicate that enforces its behaviour, whose encoding depends on the complexity of the start and stop events. Figure 5 presents those of mute (`setMuteTimer`) and disconnect propagation (`setDiscPropTimer`) timers; the remainder are omitted from the excerpt. Fact `setTimers` aggregates these predicates and forces them to hold in each instant. No particular duration is imposed on timers, so these predicates only model the possibility of expiration, and not its enforcement. Since each step does not represent any particular real-time interval, the free expiration allows for the designer to test different interleavings. Electrum has limited support for integers, which could allow for the eventual codification of real-time timers. However, we did not find that level of detail to be helpful in the kind of analyses performed.

Mute timers [10, §3.4.1.2] start when a train reports position information, and stop when a train is identified as disconnected. Thus, they may expire when a connected train fails to report information, as encoded in predicate `setMuteTimer` in Fig. 5. Wait integrity timers [10, §3.4.1.3] follow the same rationale. Notice how only the upper bound for expired timers is specified, encoding only the possibility of expiration. For timers with more complex start conditions (shadow timers [10, §3.4.1.4,§3.4.1.5] and propagation timers [10, §3.4.2]) functions were defined to collect timers for which those conditions are met. Function `DiscPropStart`, e.g., models the 3 conditions for disconnect propagation timers to start [10, §3.4.2.2]. Disconnect and integrity loss propagation timers also have complex stop conditions, so similar functions were defined, as `DiscPropStop` for the former. These sets are used to update the set of running timers by adding those starting and removing those stopping and expiring (see predicate `setDiscPropStop`). All propagation timers in version 1C have at least one stop event, applied when the VSS state machine finishes executing, so that they are only processed once [10, §3.4.2.1.2]. The last constraint on `setDiscPropTimer` removes expired timers from the succeeding state.

Lastly, the VSS state machine [10, §5.1.1.1] can be encoded, combining the inferred state of the trains and the timers, whose outcome is used to issue MAs. Fact `stateMachine` in Fig. 5 updates the state of the VSSs by enforcing the various transitions. Depending on the current state of each VSS, transition conditions are tested in an order that preserves the priorities specified in the requirements. The structure of the requirements is preserved, with a predicate defined for each alternative condition. This translation of the triggering conditions into our model's predicates is rather straightforward, especially with the aligned nomenclature for location information. As an example, conditions for transition #4 between unknown and free states are depicted. Due to the complexity (and occasional ambiguity) of these conditions, this construction process was iterative with the encoding of the operational scenarios (Section 4.1.2). Other than the issue already identified with the assumed rear end location in transition #10A, the only other condition not expressible in our model is #11A regarding the minimum safe rear end position and the distance that can be covered within the shadow train timer A. Although this did not affect the operational scenarios and the verified properties, it would be relevant to exactly identify the consequences of this omission.

4 Verification & Validation

The Electrum Analyzer provides automatic analysis procedures that can be used to validate and verify the developed models through **`run`** – generate instances for which a property holds – and **`check`** – check whether a

property holds for all instances – commands. Two alternative engines are provided, one bounded (SAT-based) and one complete (SMV-based) that can be used at different stages of development. This section describes how these were used in the development of our model.

4.1 Validation

A conceptual model must be validated against the requirements and with other relevant stakeholders. The Electrum Analyzer provides support to generate solutions to the model that satisfy provided properties, allowing for the specification and exploration of scenarios, as well as providing a graphical visualizer.

4.1.1 Scenario Visualisation

The Electrum Analyzer provides a graph visualiser for depicting the found instances, whose appearance can be customisable through themes. This is essentially an extension to the Alloy Analyzer to natively support infinite temporal traces through loopbacks. These logic-agnostic graphical instances are understandable by stakeholders without expertise in formal specification, and have previously proven to be suitable for establishing a common interpretation of the requirements [22]. We focused on providing a visualisation theme that allowed both software designers and ERTMS/ETCS domain experts to communicate through a common scheme.

The Analyzer's theme editor provides basic customisation functionalities (e.g., changing the shape, colour and border of elements). More customizations can be performed by defining additional functions in the model, whose result is calculated at static time by the visualizer. For instance, one can draw occupied VSSs differently by creating a function that retrieves all those VSSs:

```
fun occupied : set VSS {
  { v:VSS | v.state = Occupied } }
```

Given the theme customizations, the Alloy Analyzer applies a graph representation algorithm and distributes nodes among layers, a process that is oblivious of the underlying semantics of the nodes and edges. The only mechanism available to the user to change the shape of this graph is to reverse the direction of edges. In our HL3 model, this resulted in a graph that, although layered into TTDs, VSSs and trains, did not preserve the order on TTD and VSS blocks, hindering the readability of scenarios. To overcome this, we implemented a small modification of the Electrum Analyzer where information regarding totally ordered sets (`TTD` and `VSS` in HL3) is passed down to the visualizer and, when possible, used to order such elements in the same graphical layer.

The developed theme[3] depicts HL3 instances and counter-examples as the snapshot in Fig. 6 for Scenario 2. TTD sections and VSS subsections appear layered and ordered, with different colours depending on their current state (a textual label is also present). A train representation depicts (textually and graphically) its position, reporting status and EoA. Running and expired timers are also depicted. Figure 6 in particular denotes a `split` event, where a two-carriage train breaks up, one failing to report integrity and the other becoming disconnected.

4.1.2 Modelling the Operational Scenarios

Electrum specifications can be animated through **`run`** commands that, given an arbitrary desirable property and a finite scope for the declared signatures, automatically search for satisfying instances. Each signature scope denotes the maximum (or **`exactly`** the) number of elements that will be considered by the Analyzer. When performing bounded model checking, the maximum trace length that will be considered is imposed by a scope on **`Time`**. Once a solution is found, additional non-isomorphic solutions can be efficiently navigated through the Analyzer.

This functionality allows the user to quickly explore loosely defined scenarios and reason about model instances that satisfy properties of varying complexity. They were heavily used throughout the development of the model to validate each introduced feature. For instance, for an initial insight on whether jumping trains are being detected as expected (field `jumpng`), the following command was specified and executed

```
run { eventually some jumpng }
  for 8 Time, 3 Train, 3 TTD, 8 VSS
```

which will generate a trace where a jumping train is detected. Alternative solutions, with arbitrary track configurations within the scope, can then be quickly iterated, helping the user detect problematic instances.

The HL3 concept [10, §6] provides a set of operational scenarios, whose specification and animation also proved essential to validate the model during development, and acted as regression tests for subsequent changes. All 9 scenarios (and 2 variants of Scenarios 2 and 5) were encoded in Electrum, and their outcome can be consulted online.[4] Our approach to the encoding of the operational scenarios is presented below. Modulo some differences due to design decisions already presented in Section 3 and to (in our view) few inconsistencies in the reference document, we were able to animate every scenario.

[3] `http://haslab.github.io/Electrum/ertms_1C.thm`
[4] `https://github.com/haslab/Electrum/wiki/ERTMS`

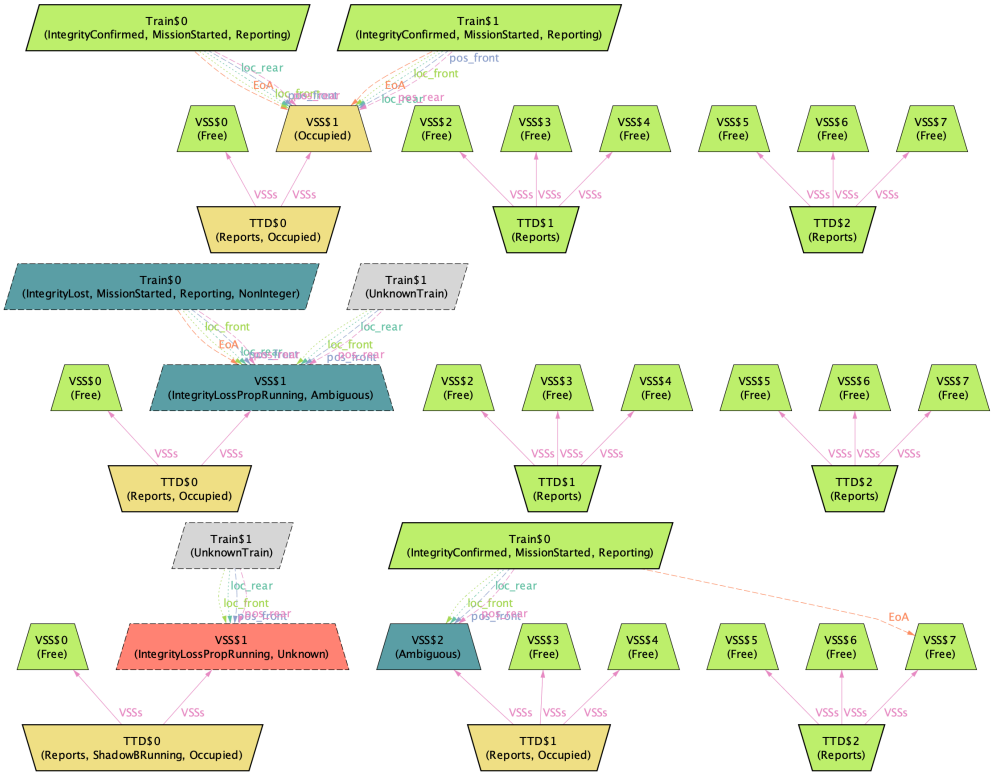

Fig. 6 The first 3 steps of the HL3 operational Scenario 2.

As already stated when discussing the architecture of our HL3 model, scenarios bound the environment and allow the VSS sub-system to evolve freely. Thus, we started by modelling as a **pred** the environment, comprised by 5 blocks restricting at each state: the track configuration (actually the same in all scenarios), the TTD reporting delays, the state of the trains, the PTD communication, the MA assignments (recall that no concrete policies are enforced) and the expiration of timers (recall that no timer duration is imposed).

Specifying tight scenarios with several steps in Electrum is verbose, since LTL does not allow the reference to concrete time instants, resulting in formulas with nested **after** operators. This was manifest when developing the HL3 model, where every scenario has at least 8 steps. This led us to explore potential language extensions to ease the specification of such scenarios, including the introduction of a new (syntactic sugar) operator: rather than `p and after (q and after r)` one can now simply write `p;q;r`. Figure 7 presents an excerpt of predicate `S2env` encoding the environment of Scenario 2 relying on LTL and this new operator.

At this point **run** commands were used to animate the scenarios and inspect the behaviour of the VSS sub-system. Since the trace length of scenarios is known, the bounded engine of the Analyzer is better suited to for this task. Results are visualized as depicted in Fig. 6 for Scenario 2. Unsatisfiability of these commands usually amounted to issues in the dynamic environment model.

```
pred S2env {
  let v11 = V/first, v12 = v11.next, v21 = v12.next ... {
  some disj t1,t2:Train {
    // track configuration
    v12 in parent[first].end and v31 in parent[last].start
    // TTD reporting
    always Reports = TTD
    // train positions
    t1.pos = v12;t1.pos = v12;t1.pos = v21;...
    always t2.pos = v12
    // PTD reporting
    split[t1,t2]
    after after after after after (historically t1 in Reporting;
                    t1 not in Reporting;always t1 in Reporting)
    t1 in IntgrtyConfirmed;t1 not in IntgrtyConfirmed;...
    // MA assignments
    t1.EoA = v12;t1.EoA = v12;always t1.EoA = v33
    t2.EoA = v12;always no t2.EoA
    // timer expiration
    after after after (historically no IntgrtyLossPropExpired;
                                  v12 = IntgrtyLossPropExpired)
    ... } } }
pred S2ok {
  let v11 = V/first, v12 = v11.next, v21 = v12.next ... {
    // final VSS state
    eventually always {
      (v11+v12).state = Unknown
      v31.state = Occupied
      (v21+v22+v23+v32+v33).state = Free }
    // timer start events
    v12 = IntgrtyLossPropRunning;v12 = IntgrtyLossPropRunning'
    D/first.next = ShadwARunning''''' } }
```

Fig. 7 Excerpt of the Scenario 2 specification.

For each scenario, a predicate denoting the expected outcome for the VSS sub-system was then specified, which tests whether timers are correctly triggered and whether the expected final state of the VSS is reached, and thus the consistency of the scenario under our model. It also tests whether the final state is stable, in the sense that no spurious state changes are triggered (recall that traces are always infinite, so a stable final state should loop into itself). Figure 7 denotes such predicate `S2ok` for Scenario 2, which can finally be simulated through the following command:

```
run { S2env and S2ok } for exactly 8 Time,
  exactly 2 Train, exactly 3 TTD, exactly 8 VSS
```

At this point the environment predicate is known to be consistent, so unsatisfiability usually amounted to issues in VSS sub-system model. Occasionally however, inconsistencies between the VSS sub-system requirements and the reference operational scenarios were detected, which are discussed in Section 5. It could also represent non-stable final states, which required longer traces to form a loop (e.g., Scenario 9, see Section 5), meaning that the trace length scope needed to be increased.

All operational scenarios have 3 TTDs, 8 VSSs and either 1 or 2 trains, so the scopes can be bound exactly in the commands. At the beginning of the development of HL3, scope **`Time`** denoted the maximum trace lengths that would be explored in bounded analysis. A scope n on **`Time`** would launch an iterative process to check traces up to n length. This is important, since the absence of a counter-example for length n does not entail its absence for some $m < n$. However, in the HL3 model we are aware of the exact number of steps that comprises each scenario, and, since this number is not particularly small (at least 8 states), the incremental iterative process encumbers the solving process. Thus, the Analyzer was adapted to support ranges or exact bounds for **`Time`**, allowing for the faster generation of scenarios.

Finally, once the scenario was stable and shown to be consistent, we encoded a **`check`** predicate to assert whether the expected behaviour of the VSS sub-system is the only one acceptable under our model:

```
check { S2env implies S2ok } for exactly 8 Time,
  exactly 2 Train, exactly 3 TTD, exactly 8 VSS
```

Counter-examples could identify under-specified transitions or timer events, but usually regarded under-specified environment restrictions on scenarios.

4.2 Verification

Proper validation increased our confidence that the model effectively abstracts the behaviour specified in the HL3 concept. The next logical step is to verify whether such model behaves as expected. However, other than the general goal of avoiding collisions, there is no explicit notion of correctness defined in [10]. Moreover, this correctness is dependent on behaviour that is outside the scope of [10], namely the policy for extending and shortening MAs, as well as how the train acts upon those MAs. As a consequence, this exercise was mainly exploratory, although we hope that these preliminary results can foment the discussion among domain experts and lead to more formally defined safety requirements for implementations of the HL3 concept.

As should be expected, without additional restrictions on the train movement in relation to the assigned MAs, no safety property would hold. After analysing counter-examples, two additional assumptions were defined (predicate `strictMove`, not shown): *i*) trains with MAs assigned always move within them, and *ii*) trains without MAs assigned or with OS MAs do not move into sections with other trains on it. Recall that in our model disconnected trains may have their MAs removed, so with `strictMove` they will act "on sight" unaware of MAs. It should be noted however, that there are (not-specified) reasons for trains to move outside the assigned MAs [10, §1.2.3.3], as in Scenario 8, where a connected train moves into VSS21, outside its assigned EoA VSS12. Although train movement were further restricted, MA assignment policies were still left under-specified.

We started by exploring properties regarding the behaviour of the VSS state machine. A reasonable correctness property is that, if communication never fails nor integrity problems occur (predicate `noProblems`, not shown), only free or occupied VSSs should occur. In fact, every VSS with a train on it should be set as occupied and the others as free. Electrum allows the definition of assertions as regular formulas, so that they can be re-used in multiple check commands. For instance, the following asserts whether VSSs with trains in it are always marked as occupied

```
assert trains_Occupied {
  (init and always (strictMove and noProblems))
    implies always Train.pos.state = Occupied }
```

where state predicate `init` encodes a sensible initial state, forcing all trains to be reporting and the VSSs to have a consistent state.

Assertions were then checked for increasing scopes using the bounded engine until a considerable level of confidence was attained. For instance, `trains_Occupied` was shown to hold up to the following scope:

```
check trains_Occupied
  for 10 Time, 8 VSS, 3 TTD, 3 Train
```

This encompasses 63 different configurations (21 tracks, with up to 3 trains). Finally, the complete model checking engine was employed, guaranteeing that the property

holds for traces of any length (the scope on **Time** is ignored in this mode), albeit with smaller scopes (5 VSSs, 2 TTDs and 2 train, 8 configurations).

Other interesting safety properties should allow for failures in communication, which necessarily involves reasoning about timers. Timers are used to avoid unnecessary state changes, but if assumed to expire instantaneously should guarantee the correct assignment of states to VSSs at all times. Our model does not impose particular timer durations, but they can be forced to expire instantaneously by declaring that the running signatures are contained in the expired ones (predicate `instTimers`, not shown). Then it would be sensible to expect, e.g., that VSSs with trains are not free:

```
assert timers_Free {
  (init and always (strictMove and instTimers) implies
    always Train.pos_frnt.state != Free }
```

Unfortunately, counter-examples are found to this property, such as the one in Fig. 8, with a train physically located in free VSS. We were unable to find sensible restrictions to train movement and MA assignment to render this property true. More complex assertions could test alternative timer durations and reason about possible interleaving issues among different types of timers.

A more general goal of HL3 and the VSS management sub-system is to preserve the safety of the trains, i.e., that no collision occur:

```
pred noCollisions {
  no disj t1,t2:Train | some t1.pos&t2.pos }
assert no_collisions {
  (init and always (strictMove and instTimers))
    implies always noCollisions }
```

Without additional restrictions on MA assignment, this property does not hold. It can be shown to be true for 2 trains if *i*) no OS MAs are assigned and *ii*) disconnected trains are only assigned MAs over free VSS sections (`MAAssigment` enforced this only for connected ones). The former is expected, since OS MAs allow trains to move freely; the latter, although seemingly sensible, does not hold on Scenario 7. Nonetheless, they are not sufficient to avoid collisions if 3 trains are considered.

5 Other Observations

5.1 Issues with the Reference Document

Validation and verification allowed us to detect possible ambiguities or under-specifications in the HL3 concept. Note that this analysis is essentially based on [10] without any *a priori* domain knowledge by the authors. Some relevant issues were identified in version 1A of the document, and have been fixed as of version 1C; they are included the discussion for the sake of completeness.

Two of these issues regarded transitions in the VSS state machine, namely #1A and #5A, that when codified as described in [9, §5.1.1.1] were inconsistent with the operational scenarios. Condition #1A triggered the transition between a free VSS into unknown whenever the parent TTD was occupied without a train located *or* without an MA assigned. Yet some scenarios did not reflect this behaviour, like Scenario 7, where VSS33 should transition to unknown since no train was located in the occupied TTD30. At the time we proposed dropping the second disjunct or converting the condition into a conjunction. The latter is now encoded in version 1C of the reference document. Transition #5A between unknown and ambiguous should be triggered whenever a train was *located* in the VSS. For the remainder transitions, "located" was assumed to denote the last known position of the train. Yet, several scenarios broke under this interpretation for #5A, like VSS22 at Scenario 4 that remained unknown even though the last reported position of the train was that VSS. A more flexible notion of location for #5A had to be considered to match the scenarios' behaviour. In version 1C of the document, several conditions were modified and the location information to be used has been clarified (namely, when memorised information should be used). This issue no longer occurs in version 1C of our model.

Another issue regarded the indefinite expiration of timers. Although [9, §3.4.2.1] stated that expired timers remained expired until the start conditions were met again, this behaviour did not seem to be followed in the operational scenarios. For instance, in Scenario 9, if the ghost propagation timer remained expired, VSSs at TTD30 would transition from free to unknown according to #1F. At the time, we did not implement indefinite expiration of timers, so that they would only be processed once. Version 1C of the reference document fixed this issue and no longer imposes indefinite expiration for propagation timers [10, §3.4.2.1].

New issues were identified in version 1C of the reference document, mainly regarding timers. First, the stop event conditions for disconnect propagation timers [10, §3.4.2.2.2] seem to be too loose: condition *b*) is triggered if a VSS becomes free, which would not allow the timer at VSS12 to expire at step 5 of Scenario 6. The same applies to the stop event conditions for integrity loss propagation timers [10, §3.4.2.4.2]: condition *a*) tests whether all trains in the VSS confirm integrity, which breaks every scenario where such timers are relevant. We believe it is more sensible to test the integrity of the trains for which that timer had started (much like stop condition *a*) of the disconnect propagation timer, that tests the connectivity of the trains for which that timer had started). Unfortunately, this still breaks Scenario 2,

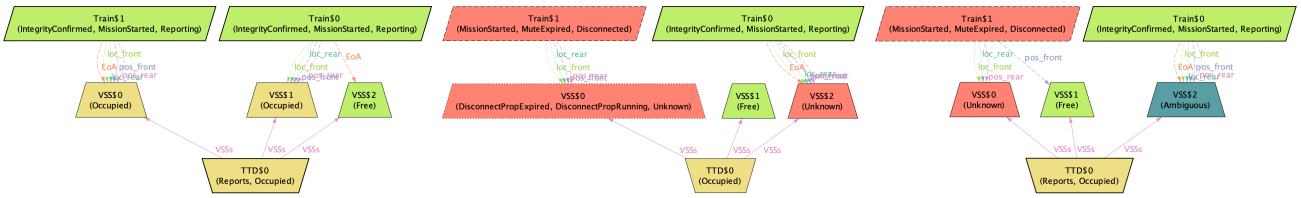

Fig. 8 Minimal counter-example for instantaneous timer expiration.

where the timer is triggered even though train 1 confirms integrity. In Scenario 3 there seems to be an inconsistency due to legacy from version 1A: the shadow timer B is said to not be start due to a condition on the rear end of the train, but as of version 1C, the start event of shadow timer B has no such condition [10, §3.4.1.5].

A (possible) issue identified when encoding the operational scenarios is that the VSS state machine may not stabilize for fixed environments. Namely, Scenario 9 does not stabilize in the specified final state: as depicted in Fig. 9, even though the environment does not change, the state of VSS21 keeps alternating between unknown and ambiguous, triggered by transition #5A, since `Train$0` never updates its rear position.

Although MA policies are not the focus of the HL3, the lack of information regarding the impact of such assignments proved confusing. For instance, when modelling version 1A, two kinds of MAs seemed relevant: FS MAs up to a specific VSS and OS MAs over the full track. Yet, version 1C uses OS MAs limited by VSSs and FS MAs with optional OS mode. Their impact was unclear to us, as they did not seem to affect the VSS sub-system requirements nor the operational scenarios. Moreover, our assumption for verifying safety properties, that connected trains respect assigned MAs, is broken in Scenario 8, although the phenomenon is not mentioned in its description. If depicting this unpredictable behaviour in a scenario was intended, it would be helpful to clearly state so in the description. In general, although the reference document makes no claim of completeness, it would be helpful for validation purposes to clarify the criteria followed in the description of the scenarios. For instance, even though the scenarios often mention when timers start running, we have found timers (mostly shadow timers) to be triggered in our model but not reported in the scenarios (e.g., Scenario 4., shadow timer A at TTD10, or Scenario 5., shadow timer B at TTD10).

5.2 Electrum Evaluation

The analysis of the HL3 under Electrum triggered improvements on the language and its Analyzer. These have already been introduced in previous sections, and aim essentially to ease the specification and animation of scenarios: *i*) the long and rich traces of the case study motivated the introduction of a new operator `;` to ease the specification of concrete traces; *ii*) the long traces also motivated the introduction of finer scopes enforced over their length, reducing the solving times by focusing on specific lengths; *iii*) and the need to present such complex scenarios in a manner understandable by all interested parties (as close as possible to that from the reference document) led us to tweak the visualizer to consider information from totally ordered atoms.

The Analyzer allowed for the automatic generation of scenarios and checking of assertions. Analyses were run in a quad-core Intel Core i5-4200U Haswell with 4GB RAM, the bounded engine relying on MiniSAT and the unbounded on nuXmv. Bounded performance of scenario generation ranged from 11s for Scenario 1 to 34s for Scenario 9. Regarding the safety properties, bounded analysis of `trains_Occupied`, e.g., took 287s for 8 VSSs, 3 TTDs and 2 trains (42 configurations), for 8 instants of time. The same property, but for 5 VSSs and 2 TTDs (8 configurations), took 514s in the unbounded engine.

Bounded model checking can sometimes have unpredictable effects for those unaccustomed with its semantics. As already reported, the infinite traces forbid deadlocks at the last state, forcing the trace to loopback into a previous state. This may lead to unexpected unsatisfiable commands and the need to extend traces (on a positive note, this also forced us to reason about the stability of the VSS state machine).

6 Comparison

6.1 Comparison with other submissions

The comparison with other HL3 submission focuses on differences of methodology, capabilities of the formalism and toolkits, and main outcomes, rather than on the interpretation and modelling of the HL3. Note that they were developed for version 1A of the reference document.

Submissions followed different modelling approaches. Hansen et al. [13] modelled HL3 in B [1] while Mammar et al. [20] and Abrial [3] relied on Event-B [2], supported by the Rodin toolkit [4]. B-related approaches require the system to be explicitly encoded as a state machine.

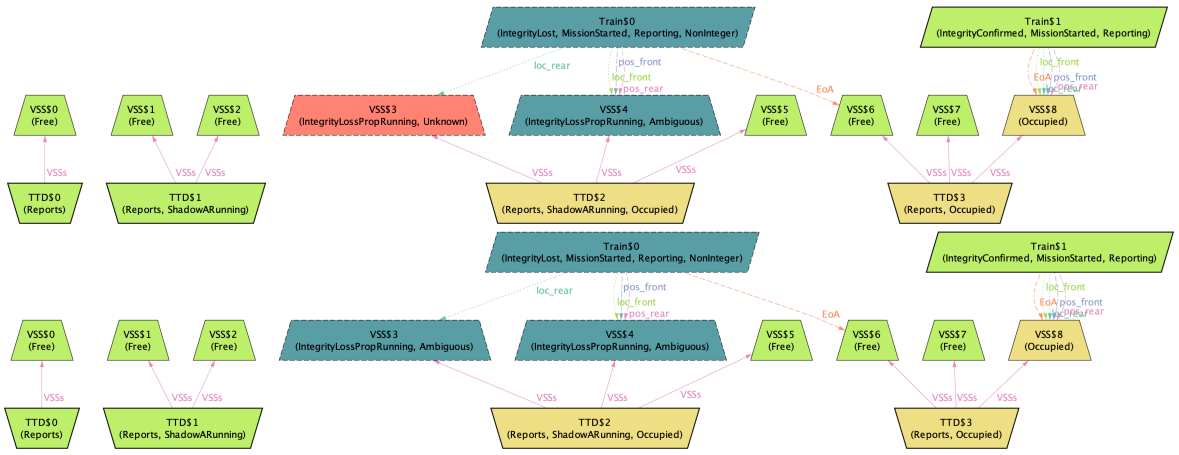

Fig. 9 Non-stabilizing loop at the end Scenario 9.

Invariants over them can then be specified and verified, with no direct support for temporal logic. Arcaini et al. [5] propose an approach based on the Spin explicit model checker [15] by modelling the HL3 in the Promela language. Models are a set of processes (state variables and statements to update them) over which assertions, or more general LTL properties, can be checked. In contrast in Electrum there is no explicit notion of state machine, which is implicitly defined by full first-order LTL restrictions. Likewise, the specification of properties can rely on the full Electrum language, although we mostly verified safety properties.

Other authors relied on higher-level, graphical modelling languages, which were then translated into formal models, arguing that they promote a better understanding of the model between the stakeholders. Dghaym et al. [8] diagrammatically modelled HL3 in iUML-B [24, 23] class diagrams and state machines in Rodin, which are converted into Event-B. Fotso et al. [12] encode the HL3 in a higher-level formalism, SysML/KAOS [21], through goal diagrams and domain models. These are translated into B System, a variant of Event-B supported by Atelier-B[5], creating skeletons for events.

Our model encodes arbitrary track configurations (within a given scope for VSSs and TTDs), supporting the analysis of TTDs with different number of sub-blocks. As far as we can tell, the other approaches encode track partition as constants of the model [3, 20, 8, 12], in the initial state of the state machine [5] or as input configurations [13]. This makes it harder to verify properties for all possible configurations of track partitions.

The kinds of properties verified varied considerably. In [13] invariants regarding train status and location were specified and model checked by ProB [18], which generates counter-examples to broken properties. In [3], all invariants relevant for a simplified HL3 model, that regard train status and VSS states, were proved using Rodin, 70% automatically and the remainder interactively. [20] focused on no collision properties and determinism of VSS state machine. ProB was used to quickly find counter-examples and Rodin to carry out proofs. In [8] the main addressed safety property was also the absence of collisions, with most of the proof obligations generated by Rodin automatically discharged. In [12] Rodin is used to (automatically and interactively) verify invariants and the proof obligations generated by the translation from SysML/KAOS. They conclude that collisions may not be avoided for disconnected trains since they may move freely. In [5] the safety properties were verified by model checking in Spin: trains do not collide and do not move beyond the assigned MAs. The authors also show that the VSS state machine is not deterministic. The Electrum Analyzer only supports model checking, and not theorem proving.

Our approach allows the user to animate models to quickly identify problematic issues by generating instances for which certain restrictions (in full first-order LTL) hold. Other frameworks provide animation functionalities that have been used by the other submissions, namely ProB for Event-B approaches [12, 8, 20] (although few details are provided on how cumbersome or helpful the process was), and Spin for the Promela models [5]. Unlike the Electrum Analyzer, the animators provided guided simulation where the user selects the next state interactively. Although this can prove infeasible for larger models [5], we believe that such functionality is helpful in early model development stages, and is one of our current focus of research for Electrum. The Analyzer does provide an alternative scenario exploration functionality by allowing the user to randomly iterate over alternative, non-isomorphic instances.

[5] https://www.atelierb.eu

Although simulation is helpful for scenario exploration, there is often the need to encode and replay specific scenarios, e.g., when reference examples are provided, or to be used as regression tests. In particular, we, as others, relied heavily on the inspection and codification of the provided operational scenarios. In [20], ProB was used to animate the operational scenarios, but no information is provided regarding the process and how cumbersome it was. In [13] all operational scenarios were encoded as environment models, but again no details regarding this process are provided. In [5] all operational scenarios were animated, but since the authors could not rely on the guided simulation functionalities of Spin, an alternative approach was followed. The state of every step in the scenarios was encoded in Promela, and the model transitions were adapted to read specific states when a scenario was specified as an input parameter. In Electrum scenarios can be directly encoded as axioms using full LTL formulas, without imposing any change on the remainder of the model. The use of declarative LTL, we argue, also results in more readable scenarios than explicitly specifying the complete state of each step. Issues in the reference document were detected in this process in [5] and in [13], including in the latter the issue with transition #1A we also identified.

The ability to graphically visualize model instances promotes the communication between the interested parties. In [5] standard output printing instructions were added to the model definition to inspect scenarios. ProB supports custom visualizers, and in [13] the authors developed one such plug-in that was used both for validation and in the executable plug-in, which proved helpful to communicate among the team and domain experts. Although we believe Alloy's / Electrum's visualization functionalities to be quite useful, the ability to further customize visualization in complex projects would be helpful. We are currently studying this topic [6].

An issue not addressed by this work is how to obtain an executable component from our model. B-based approaches are naturally better-suited for this, as model refinement is a key feature of the formalism, which is still an unexplored issue in Alloy/Electrum. An alternative approach is followed in [13], where ProB's Java API is used to create an executable component that was integrated into a real *Radio Block Centre* system that processes trackside information and issues MAs. It would be interesting to assess whether Electrum's Java API would be feasible this purpose, performance-wise.

6.2 Comparison with Alloy

Being an extension to Alloy, it is important to compare the verbosity and readability of Electrum models with those developed in normal Alloy. Thus, a similar encoding of the version 1A of the HL3 concept was developed in Alloy as well[6], which, given the complexity of the case study, enabled us to clearly picture the cons and pros of the two languages. The static structure of the system is identical in either language. In Alloy however, time, mutability and dynamic properties must be explicitly modelled. This requires the modelling of traces, by declaring a signature `Time` and imposing a total order on it, and the conversion of all variable signatures and fields to a state idiom [17], where, e.g., field `pos_frnt` would be declared with type `VSS one → Time`. Temporal formulas must also explicitly quantify over time instants. For instance assertion `no_collisions` could take the shape:

```
pred noCollisions[t:Time] {
  no disj t1,t2:Train | some t1.pos.t&t2.pos.t }
assert no_collisions {
  (init[first] and all t:Time | strictMove[t])
    implies all t:Time | noCollisions[t] }
```

The tradeoff is that Electrum does not allow quantification over time instants. For instance, to retrieve the last reported train location in Alloy one can retrieve the last state `s1` in which a train reported, treat it as a first-level entity throughout relational formulas and expressions, and use it to query the state of the system at that state:

```
let s1 = max[s.*prev&t.Reporting] | t.pos_frnt.s1
```

Various alternative Electrum encodings can be employed for the same purpose, using LTL with past operators, or introducing additional variables in the model. The latter was followed by introducing field `mem_fr` to register the location whenever a train is `Reporting` (fact `memoryUpdate`), which is then retrieved by function `locatedFrnt` (Fig. 4). An alternative to introducing new fields would be to rely on the **since** past operator directly in `locatedFrnt`:

```
{ v:VSS | (t not in Reporting) since
            (t in Reporting and v = t.pos_frnt) }
```

Other expressions also require a different formulation in Electrum. For instance, evaluating a field `r` over every instant except `t` can be encoded in Alloy as `r.(Time-t)`, while in Electrum it must formalized as an LTL formula.

7 Conclusions

This paper reports on the modelling, validation and verification of the Hybrid ERTMS/ETCS Level 3 concept, version 1C, in Electrum, extending and improving previous work on version 1A [7]. Electrum proved well-suited to model most relevant features of the HL3, as well as the provided operational scenarios. Its Analyzer

[6] `http://haslab.github.io/Electrum/ertms.als`

allowed the automatic animation of scenarios and verification of simple safety properties. We believe that the visualization of the scenarios/counter-examples with the customized theme also promotes its understanding by different stakeholders.

The complexity of the HL3 concept has tested Electrum and its Analyzer to their limits, allowing us to fully explore their potential and identify possible improvements and future lines of research. Some improvements (minor changes to the visualizer, a new temporal operator for formulas over traces, more control on the scope of trace lengths) were already implemented throughout the development of the HL3 model. The proposed model could still be further developed to allow reasoning about some HL3 aspects that were abstracted in the current version, including train lengths and forcing the stepwise VSS state update, although we expect them to have a considerable toll on performance.

The predicates encoding the operational scenarios proved to be quite verbose, so we are currently exploring potential extensions to the language to address this, including variants of temporal logic with support for intervals that would allow the definition of properties over ranges of steps. It should be noted however that Electrum's (and Alloy for that matter) greatest strength is on the exploration of scenarios, and not the specification of fixed instances. Although we advocate that the current graphical feedback can be understood by stakeholders without background on formal specification, we also believe that there is room for improvement. We are currently working on techniques specifically tailored for the visualization and animation of traces [6]. Another line of research that we are currently pursuing, and that would prove helpful for modelling and validating systems of this complexity, aims to provide Electrum with guided simulation functionalities.

We hope that this preliminary work can help clarify some ambiguities in the HL3 concept and motivate the ERTMS/ETCS community to explore the potential of formal specification and analysis methodologies.

References

1. J. Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.
2. J. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
3. J. Abrial. The ABZ-2018 case study with Event-B. In *ABZ*, volume 10817 of *LNCS*, pages 322–337. Springer, 2018.
4. J. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.
5. P. Arcaini, P. Jezek, and J. Kofron. Modelling the hybrid ERTMS/ETCS level 3 case study in Spin. In *ABZ*, volume 10817 of *LNCS*, pages 277–291. Springer, 2018.
6. R. Couto, J. C. Campos, N. Macedo, and A. Cunha. Improving the visualization of Alloy instances. In *F-IDE@FLoC*, volume 284 of *EPTCS*, pages 37–52, 2018.
7. A. Cunha and N. Macedo. Validating the hybrid ERTMS/ETCS Level 3 concept with Electrum. In *ABZ*, volume 10817 of *LNCS*, pages 307–321. Springer, 2018.
8. D. Dghaym, M. Poppleton, and C. Snook. Diagram-led formal modelling using iUML-B for hybrid ERTMS level 3. In *ABZ*, volume 10817 of *LNCS*, pages 338–352. Springer, 2018.
9. EEIG ERTMS Users Group. Hybrid ERTMS/ETCS level 3 – principles. Available at `http://www.ertms.be/sites/default/files/2018-03/16E0421A_HL3.pdf` 16E042, version 1A, 2017.
10. EEIG ERTMS Users Group. Hybrid ERTMS/ETCS level 3 – principles. Available at `https://ertms.be/sites/default/files/2018-07/16E0421C_HL3-clean.pdf` 16E042, version 1C, 2018.
11. ERA, UNISIG, and EEIG ERTMS Users Group. Glossary of unisig terms and abbreviations. Available at `https://www.era.europa.eu/filebrowser/download/492_en` SUBSET-023, issue 3.3.0, 2016.
12. S. Fotso, M. Frappier, R. Laleau, and A. Mammar. Modeling the hybrid ERTMS/ETCS level 3 standard using a formal requirements engineering approach. In *ABZ*, volume 10817 of *LNCS*, pages 262–276. Springer, 2018.
13. D. Hansen, M. Leuschel, D. Schneider, S. Krings, P. Körner, T. Naulin, N. Nayeri, and F. Skowron. Using a formal B model at runtime in a demonstration of the ETCS hybrid level 3 concept with real trains. In *ABZ*, volume 10817 of *LNCS*, pages 292–306. Springer, 2018.
14. T. S. Hoang, M. Butler, and K. Reichl. The hybrid ERTMS/ETCS level 3 case study. In *ABZ*, volume 10817 of *LNCS*, pages 251–261. Springer, 2018.
15. G. J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.
16. INESC TEC and ONERA. Electrum Analyzer, v1.0. Available under the MIT License at `https://github.com/haslab/Electrum/releases/tag/v1.0`, 2018.
17. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2012.
18. M. Leuschel and M. Butler. ProB: A model checker for B. In *FME*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.
19. N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In *SIGSOFT FSE*, pages 373–383. ACM, 2016.
20. A. Mammar, M. Frappier, S. Fotso, and R. Laleau. An Event-B model of the hybrid ERTMS/ETCS level 3 standard. In *ABZ*, volume 10817 of *LNCS*, pages 353–366. Springer, 2018.
21. A. Mammar and R. Laleau. On the use of domain and system knowledge modeling in goal-based Event-B specifications. In *ISoLA*, volume 9952 of *LNCS*, pages 325–339, 2016.
22. J. M. Moreira, A. Cunha, and N. Macedo. An ORCID based synchronization framework for a national CRIS ecosystem. *F1000Research*, 4(181), 2015.
23. C. Snook. iUML-B statemachines. In *Proceedings of the Rodin Workshop 2014*, pages 29–30, 2014. Available at `http://eprints.soton.ac.uk/365301/`.
24. C. Snook and M. Butler. UML-B: formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.