Validating the Hybrid ERTMS/ETCS Level 3 Concept with Electrum

Alcino Cunha and Nuno Macedo

INESC TEC & Universidade do Minho, Portugal

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 example 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. The Analyzer depicts scenarios (and counter-examples) in a graphical notation that is logic-agnostic, making them understandable for stakeholders without expertise in formal specification.

1 Introduction

The *European Rail Traffic Management System* (ERTMS) is a system of standards for management and interoperation of signalling for railways by the EU, that aims to replace the various national systems with a seamless European railway system[1]. The *European Train Control System* (ETCS), the ERTMS control command part, defines 3 levels of signalling that a system can operate on, depending on the trackside equipment used, how the on-board systems communicate with the trackside, and on which functions are processed on-board or by the trackside. In Level 3, positive train detection (PTD) information, including the train position and integrity information, is detected and reported by the on-board system directly to the trackside, which, based on logical rather than physical track block sections, decides whether it is safe to issue movement authorities (MA), reporting them back to the on-board system via radio. By removing the need for physical trackside detection, the implementation cost is reduced, while the use of virtual blocks allows for arbitrarily small sections, improving the performance and adaptability of the system.

For Level 3 to be feasible, PTD information must be reliable and the communication between the on-board and trackside systems guaranteed at all times. These pre-conditions are not easily met, which has led to the proposal of a Hybrid Level 3 concept [2], that combines PTD information with limited trackside detection. These trackside train detection sections (TTD) are then broken into smaller virtual subsections (VSS). Each of these VSSs, besides being identified as free or

[1] `http://www.ertms.net/`

occupied, may also become ambiguous or unknown, whenever discrepancies in the information are detected. This allows for trains with non-ideal equipment or with communication problems to still use the line, albeit below full capacity.

This paper reports on the modelling and subsequent validation and verification of the Hybrid ERTMS/ETCS Level 3 (HL3) concept in Electrum [5], and was developed as an answer to the ABZ 2018 call for case study contributions. Electrum is a lightweight formal specification language that extends Alloy [4] with mutable relations and temporal logic 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 LTL properties. Its Analyzer [3] provides support for both bounded (through SAT likewise Alloy) and unbounded (through SMV) model checking, whose solutions (or counter-examples) are presented back to the user in a unified graphical interface.

The resulting HL3 model, as well as relevant design decisions, are presented in Section 2. Electrum concepts are presented as needed. Section 3 describes how the model was validated using the Analyzer, including the encoding of the operational scenarios, while Section 4 explores some desirable safety properties that can be automatically verified. Section 5 discusses the results and some identified challenges, and Section 6 points directions to future work. It should be noted that the authors had no *a priori* domain knowledge, and that this work was mainly based on the provided "*Principles*" document for the HL3 [2].

2 Modelling

The section presents an Electrum model for the HL3 concept, which is available online[2]. 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 the HL3, 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. Relevant design decisions regarding such issues 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 [5].

2.1 Static Structural Components

In Electrum, likewise Alloy, structure is introduced through the declaration of *signatures*, that represent sets of uninterpreted atoms, and *fields*, that create relationships between multiple atoms. In Electrum such signatures may either be static (by default) or variable (those marked as **var**), and can be restricted by

[2] `http://haslab.github.io/Electrum/ertms.ele`

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

enum State
  { Unknown, Free, Ambiguous, Occupied }

sig VSS {
  var state   : one State,
  var jumping : lone Train
}

sig TTD {
  start : VSS,
  end   : VSS,
} { end.gte[start] }

fact trackSections {
  all ttd:TTD-D/last |
    ttd.end.V/next = (ttd.D/next).start
  first.start = V/first and last.end = V/last
}

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

```
sig Train {
  var pos_front : one VSS,
  var pos_rear  : one VSS,
  var MA        : one VSS
}
var sig connected in Train {}
var sig report_front,report_rear in Train {}

fun mute : set Train {
  Train-(report_rear+report_front)
}
fun disintegrated : set Train {
  report_front - report_rear
}

fun MAs[t:Train] : set VSS {
  knownRear[tr].*V/next & (tr.MA).*V/prev
}

fun knownFront[t:Train] : one VSS {
  { v:VSS | t not in report_front since
    (t in report_front and v = t.pos_front) }
}
...
fun occupied : set TTD {
  { d : TTD |
    some VSSs[d]&Train.(pos_rear+pos_front) }
}
```

Fig. 1. Excerpt of the structure of the HL3 model.

simple multiplicity constraints. Static signatures represent the possible configurations on which a system can act, and although they can still be loosely defined and solved during the analysis process, they stay frozen throughout the evolution of the system. In the HL3 model, as shown in Fig. 1, these represent the available trains (signature `Train`) and the valid configurations of trackside train detection sections (signature `TTD`) and virtual subsections (signature `VSS`). Tracks are simply comprised by discrete sequences of VSS atoms, which in Electrum can be achieved by imposing a total order. Fields `start` and `end` simply register exactly **`one`** `VSS` in which a `TTD` starts and ends, respectively. This representation does not consider any particular dimension of the blocks or trains, which essentially affects reasoning about the minimum safe rear end position that occurs in transition #11A and the start event of the ghost propagation timer [2].

Relational expressions combine such signatures and fields (and other constants) using standard relational operators like union (`+`), intersection (`&`), difference (`-`), join (`.`) or the binary converse (~), or with transitive (`^`) or reflexive-transitive (∗) closure operators. Primed expressions refer to their value in the succeeding state. Relational expressions can also be constructed by comprehension. Primitive relational formulas are either inclusion (**`in`**) or equality (`=`) tests, or basic multiplicity tests, which can be combined through common Boolean operators, first-order quantifications or future and past LTL operators. Such formulas can be imposed as axioms that always hold in a model through *facts* such as `trackSections`, which universally quantifies over `TTD` elements to guarantee that the complete track is partitioned. *Functions* and *predicates* can be used for reusable expressions and formulas, respectively. For instance, function `VSSs` calculates all the subsections of a `TTD` through transitive closure operations, possible due to the imposed total order on `VSS`. This declarative definition allows for the analysis of properties over every valid track partition within a finite universe, a cumbersome task in languages without support for first-order logic.

2.2 Dynamic Structural Components

The dynamic structure of a model consists of the mutable elements of the system, those whose state evolves in time. In Electrum such signatures and fields are declared as **`var`**. In the HL3 model, as depicted in Fig. 1, these regard the physical state of the trains and the state of the trackside and the on-board systems.

Each train has an exact physical position (not necessarily known by the trackside) for its front and rear ends, represented by variable fields `pos_front` and `pos_rear`, that point to exactly **`one`** `VSS` at each time. Variable sub-signatures are used to represent the state of the PTD communication between each train and the trackside. Variable signature `connected` represents trains connected to the trackside at each instant, which will be modified by start (SoM) and end of mission (EoM) events, while `report_front` and `report_rear` denote which trains reported front and rear information at that instant, respectively. Dynamic auxiliary function `mute` identifies all trains not communicating in each instant, while function `disintegrated` identifies trains that failed to report their integrity (no rear report). Other auxiliary functions combine this information to retrieve the currently known information about a train. For instance, `knownFront` retrieves the last reported front position of a train using the past operator **`since`**.

Besides PTD information, an optimization is implemented in HL3 to detect the position of trains transitioning between TTDs, in order to avoid delays due to "jumping trains" [2, p. 12]. This information is assumed to exist by the VSS state machine (namely transition #2B). The field `jumping` on VSSs registers such occurrences and is also used when retrieving train position information; its content is fixed by a fact omitted from the excerpt.

HL3 proposes a state machine for VSSs, that combines the TTD and PTD information to determine the current state of a VSS, which is then used to issue MAs. This is encoded by field `state` that at each instant assigns to each VSS an element from the enumeration `State`. Unlike PTD information, the state of TTDs is considered safe: if there is a train located within it, it is reported as being occupied. Although this communication may have delays, we have opted to make it instantaneous and exact, as defined by function `occupied`. Implementing such delay would be straight-forward, by encoding an action that denotes whether TTD information has been received in each state, likewise the train PTD reporting predicates that will be presented in Section 2.3. However, this would increase the complexity of the model considerably and have little impact in the behaviour specified in [2] (it slightly affects Scenario 5 as there is no delay on detecting the free state of TTD20). Finally, the trackside system registers the end of the

```
var sig disconnect_ptimer in VSS {}
var sig integrity_loss_ptimer in VSS {}

var sig shadow_timer_A in TTD {}
var sig shadow_timer_B in TTD {}
var sig ghost_ptimer in TTD {}

var sig mute_timer in Train {}
var sig integrity_timer in Train {}

pred set_mute_timer {
  mute_timer in mute
}
pred set_integrity_timer {
  integrity_timer in disintegrated
}
```

```
pred set_shadow_timer_A {
  shadow_timer_A in start_shadow_timer_A
}
fun start_shadow_timer_A : set TTD {
  { ttd : TTD | once {
      previous ttd in occupied
      ttd not in occupied
      previous ttd.end.state = Ambiguous } }
}
...
pred set_timers {
  set_mute_timer and set_disconnect_ptimer
  set_integrity_timer and set_integrity_loss_ptimer
  set_shadow_timer_A and set_shadow_timer_B
  set_ghost_ptimer
}
```

Fig. 2. Excerpt of the timers specification in the HL3 model.

current MA for each train. All the VSSs that comprise a train's MA are calculated by the dynamic functions MAs using transitive closure operators.

To avoid performance deterioration due to communication fluctuations, HL3 implements a set of timers to avoid unnecessary state transitions. Each of these timers has start and end events, and is assigned to either a VSS, a TTD or a train. To model whether such timers have expired, for each of these elements variable sub-signatures were introduced, as depicted in Fig. 2. For instance, variable sub-signature mute_timer contains at each instant the trains for which that timer is expired. A predicate for each type of timer denotes whether the start conditions have been met. For those with dual start and stop events, this is straight-forward. For instance, a mute_timer may be triggered if a train is mute. Other timers, like the shown shadow_timer_A, must query over every previous state whether the start condition was met using the past operator **once**. Predicate set_timers aggregates all these predicates. The reference document states that, once expired, timers remain so until the start conditions are met again [2, p. 14]. Yet, this behaviour renders some of the scenarios inconsistent (see Section 3.3), so we have opted not to implement it. No particular time duration was imposed on timers, so only the possibility of expiration is modelled, 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, during analysis these are translated into their bitwise representation so that they can be handled by the SAT solvers, which encumbers the process for complex integer expressions or larger integer values.

2.3 System Evolution

The system evolves as the trains move and report PTD information, and the trackside updates the states of the VSSs and the trains' MAs. Actions that model this behaviour can easily be represented in Electrum as declarative predicates

```
pred move [t:Train] {
  t.pos_front' in t.pos_front.(iden+V/next)
  t.pos_rear' in t.pos_front'.(iden+V/prev)
  t.pos_rear' in t.pos_rear.(iden+V/next)
  { t in connected
    t in report_rear' => t in report_front'
  } or {
    t not in report_front'
    t not in report_rear' }
  t in connected iff t in connected'
}
```

```
pred som[t:Train] {
  t not in connected
  connected' = connected + t
  report_rear' = report_rear + t
  report_front' = report_front + t
  pos_front' = pos_front
  pos_rear' = pos_rear
}
```

Fig. 3. Excerpt of the train evolution of the HL3 model.

```
pred states[vss:VSS] {
  vss.state' = (
    n01[vss]   =>  Unknown else
    n02[vss]   =>  Occupied else
    n03[vss]   =>  Ambiguous else
    n04[vss]   =>  Free else
    n12[vss]   =>  Occupied else
    n05[vss]   =>  Ambiguous else
    n06[vss]   =>  Free else
    n07[vss]   =>  Unknown else
    n08[vss]   =>  Ambiguous else
    n09[vss]   =>  Free else
    n10[vss]   =>  Unknown else
    n11[vss]   =>  Occupied else
                   vss.state )
}
```

```
pred n09 [v:VSS] {
  v.state = Ambiguous
  after (n09A[v] or n09B[v])
}
pred n09A [v:VSS] {
  parent[v] not in occupied
}
pred n09B [v:VSS] {
  some t:Train {
    t not in disintegrated
    v not in knownLoc[t]
    previous v not in knownLoc[t]
    parent[v] not in shadow_timer_A
    parent[v] in start_shadow_timer_A }
}
```

Fig. 4. Excerpt of the VSS state machine specification in the HL3 model.

that relate the current state of variable elements with the succeeding one using primed expressions. More advanced actions may freely use LTL operators.

A train in the developed model can be updated by 4 events, some of which are presented in Fig. 3. SoM (`som`) and EoM (`eom`) actions simply connect or disconnect a train to the trackside. A `split` action models the breaking up of a train into two, affecting its integrity. A two-carriage train is modelled by two trains that have had exactly the same state up to that point; during break up, the front one will fail to report the rear position, resulting in lost integrity, and the rear one will be disconnected from the trackside. Finally, the `move` action updates the physical position of the train and may or not report PTD information to the trackside. To keep the evolution of the system manageable, the 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, although these restrictions could easily be relaxed. A disconnected train never reports to the trackside, while connected ones may or not do so; reports lacking rear information will model integrity loss events. Although MA policies are beyond the HL3 concept, it is assumed that trains may move outside assigned MA for operational reasons [2, p. 6]. Our model assumes that a connected train moves within its MA, while disconnected ones may disregard it. Notice that most of these actions are encoded as declarative

```
pred MAs {
  all t:connected-mute_timer |
    t.MA' = t.MA or (knownFront[t].*next&t.MA'.*prev).state = Free or after OS[t]
  all t:(Train-connected)+mute_timer |
    t.MA' = t.MA or t.MA' = knownFront[t]
}
...
fact trace {
  always {
    set_timers and MAs and all v:VSS | states[v]
    (all t:Train | move[t]) or (some t1,t2:Train | split[t1,t2] or som[t1] or eom[t1]) }
}
```

Fig. 5. Excerpt of the MA assignment and trace specification in the HL3 model.

predicates that allow for a range of behaviours at each instant. For simplicity purposes, all trains are assumed to be in the track at all times (multiplicity `one` on positions), so trains may not enter or leave the track. Modelling such behaviour is easily done by creating additional "dummy" VSSs at the beginning or end of the track, as in Scenarios 8 and 9.

When processing PTD reports, [2, p. 11] assumes that the front and rear end reports are independent events, with the front one always being processed first in case of simultaneous reports. Forcing this behaviour at all times would however double the number of steps in the generated traces, which would possibly encumber the solving process. Thus, besides the independent processing of front information (represented by reports without rear information), our model also allows the the simultaneous processing of front and rear reports. In fact, in the operational scenarios, PTD reports are collapsed into a single step, and the only scenario where this phenomenon is relevant is Scenario 9; in this case the reporting event was forced to be split into two steps in its encoding, the first missing rear information. Lastly, there are 3 events in [2] related to the integrity of the train that trigger the same VSS state transitions (#7B and #8A) and the integrity loss propagation timer (a train reports lost integrity, changed train length or its wait integrity timer expired); as these always occur in conjunction, they were abstracted into a single condition where the train fails to report the rear information, which simplified the model without affecting the overall behaviour.

Predicate `states` in Fig. 4 updates the state of the VSSs by encoding the state machine defined in [2, p. 6]. Depending on the current state of each VSS and on the available PTD and TTD information, each transition condition is tested in an order that preserves the imposed priorities. As an example, the condition for transition #9 between ambiguous and free states is depicted in Fig. 4. Due to the complexity (and occasional ambiguity) of these conditions, this construction process was iterative with the encoding of the operational scenarios (Section 3.2). Section 3.3 discusses some potential issues detected in [2] in this process.

The assignment of MAs is outside the scope of the HL3 concept [2], but the validation of the model requires that some reasonable, even if loose, policy is encoded. Its declarative definition (predicate `MAs` at Fig. 5) allows for alternative behaviours. For connected trains, either the MA remains unchanged or is updated

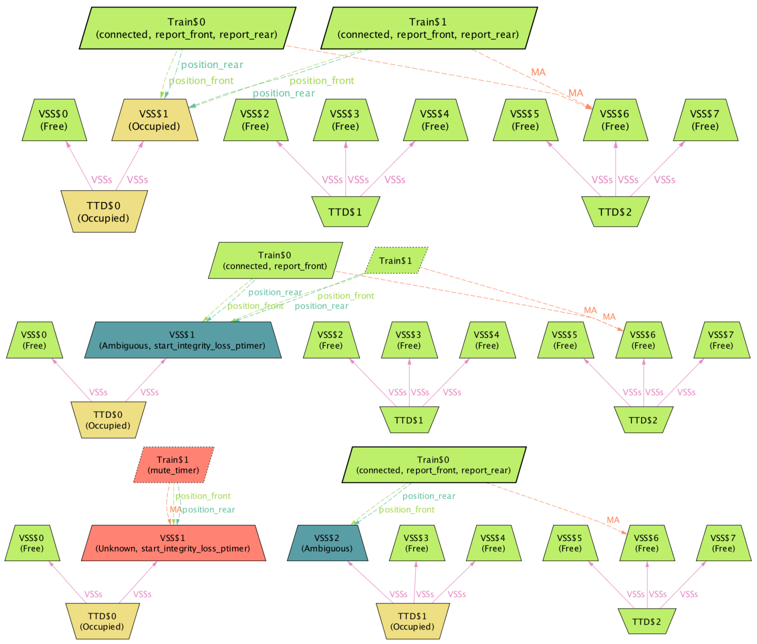

Fig. 6. 3 succeeding steps of the HL3 operational Scenario 2.

to a VSS that is only separated from the front end of the train by free VSSs. To model the on-sight (OS) operational mode, that gives full privileges to the driver, the MA may also be set to the last VSS of the track (used in Scenarios 6 and 8). For disconnected trains, either the MA is preserved, or removed altogether (by assigning it the currently known position of the train).

All the actions are enforced through a fact `trace` (Fig. 5) that guarantees that all solution traces are created from the application of these actions. These usually encode interleaving semantics of actions, but since this would lead to an explosion of steps in each trace, we allow all trains to move in each step.

3 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 logic-agnostic graphical visualizer.

3.1 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 for stakeholders without expertise in formal specification, and have previously proven to be suitable for establishing a common interpretation of the requirements [7]. 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 different signatures and fields). Additional customizations can be performed by defining functions that return sets of elements, whose result is calculated at static time by the visualizer. This enables, for instance, drawing a VSS according to its current state, by creating functions that for each state retrieves, by comprehension, VSSs elements for which that state holds, e.g., `fun occupied : set VSS {{vss:VSS | vss.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.

Using the developed theme[3], the appearance of HL3 instances and counter-examples in the Electrum Analyzer is that of the snapshot in Fig. 6 for the operational Scenario 2. Both TTD sections and VSS subsection 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 MA. Expired timers are also depicted. Figure 6 in particular denotes a `split` event, where two trains with a shared state break up, leaving one disconnected (`Train$0`) and the other moving forward.

3.2 Modelling the Operational Scenarios

Electrum specifications can be animated through **`run`** commands that, given a 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.

The HL3 concept [2] provides a set of operational scenarios that proved essential to validate the model during development. All 9 scenarios were encoded

[3] `http://haslab.github.io/Electrum/ertms.thm`

```
pred S2 {
  let v11 = V/first, v12 = v11.next, v21 = v12.next, ... {
  some disj t1,t2:Train {
    split[t1,t2]
    always t1.MA = v32
    t1 in report_front;t1 in report_front;...
    t1 in report_rear and after (t1 not in report_rear and after (...))
    t1.pos_front = v12;t1.pos_front = v12;...
    t1.pos_rear = v12 and after (t1.pos_rear = v12 and after (...))
    ... } }
}
```

Fig. 7. Excerpt of the Scenario 2 specification in the HL3 model.

as predicates in Electrum in order to guarantee that our model was not over-constrained, and were used as regression tests for any succeeding modifications. These can be automatically generated by the bounded model checking procedures of the Analyzer through the **run** commands available in the provided Electrum model. Using the provided theme, these can visualized in a style similar to the one presented in Fig. 6 for Scenario 2. The outcome of all 9 scenarios can be consulted in Electrum's website[4]. Some inconsistencies between the VSS state machine and the operational scenarios were also detected during this process, which are discussed in Section 3.3.

Specifying concrete instances with several steps in Electrum is verbose, since LTL does not allow the reference to concrete time instants, requiring the creation of 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 help specifying such scenarios, including the introduction of a new operator that acts as syntactic sugar during the specification of the traces: rather than `p and after (q and after r)` one can now simply write `p;q;r`. Figure 7 presents an excerpt of the predicate encoding Scenario 2, with rear information encoded with standard LTL operators and front information with the new operator. Running this predicate, which results in the trace depicted in Fig. 6, can be done through the following command:

```
run S2 for 8 Time, exactly 2 Train, exactly 3 TTD, exactly 8 VSS
```

All operational scenarios have 3 TTD sections and 8 VSS subsections 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 by the bounded analysis procedures, such that a scope n on **Time** would launch an iterative process where traces up to n are checked. 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

[4] https://github.com/haslab/Electrum/wiki/ERTMS

to support ranges or exact bounds for trace lengths, allowing for the faster generation of scenarios. It should also be noted that according to bounded model checking semantics [1], evidences for **always** constraints (or counter-examples to **eventually** ones) require infinite traces, represented as a finite prefix that loops back into itself. Thus, scenarios must not deadlock at the last state, but somehow loop back into a previous state. This is not possible for every state (e.g., Scenario 9), meaning that the trace length scope may need to be increased.

Notice that the scenario predicates do not completely fix the states. Instead, they focus on establishing the movement of the train (as well as some timer and MA events) and leave the VSS state machine act freely, whose state will be solved by the Analyzer.

Electrum is also useful to explore scenarios with looser restrictions, when the user wants to reason about model instances that satisfy certain properties. For instance, to explore whether the existence of jumping trains is problematic (recall that field `jumping` registers the occurrence of jumping trains) one can simply run:

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

Alternative solutions, with arbitrary track configurations within the scope, can then be quickly iterated, helping the user detect problematic instances.

3.3 Possible Issues with the HL3 Concept

Model validation allowed us to detect possible ambiguities or under-specifications in the HL3 concept. Note that this analysis is essentially based on [2] without any *a priori* domain knowledge. Two of these issues regard the VSS state machine triggering conditions, namely #1A and #5A, that when codified as described in the document result in a behaviour that does not match that of the operational scenarios. Condition #1A triggers the transition between a free VSS into unknown whenever the parent TTD is occupied without a train located *or* without MA assigned. Yet some scenarios do not reflect this behaviour, like Scenario 7, where VSS33 should transition to unknown since no train is located in the occupied TTD30. Removing the second disjunct (or converting the condition into a conjunction) results in the expected behaviour. Transition #5A between unknown and ambiguous should be triggered whenever a train is *located* in the VSS. For the remainder transitions, “located” was assumed to denote the last known position of the train. Yet, several scenarios break under this interpretation for #5A, like VSS22 at Scenario 4 that remains unknown even though the last reported position of the train was that VSS. Only considering trains reporting to be in that VSS in that instant matches the scenarios’ behaviour.

Another issue regards the indefinite expiration of timers. Although [2, p. 14] states that expired timers remain expired until the start conditions are met again, this behaviour does not seem to be followed in the operational scenarios. For instance, in Scenario 9, if the ghost propagation timer remains expired, VSSs at TTD30 should transition from free to unknown according to #1F.

4 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. Similar to the **run** commands, **check** commands in Electrum instruct the Analyzer to search for instances that break a certain assertion within a fixed scope. However, there is no explicit notion of correctness defined in [2]. Moreover, this correctness is dependent on behaviour that is outside the scope of [2], 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.

A reasonable correctness property is that, if PTD communication never fails and the integrity of the trains is never compromised, then no states other than free or occupied are assigned to the VSSs. In fact, it should be the case that every VSS with a train on it is set as occupied and the others as free. Recall that we had already imposed two (reasonable, in our perspective) assumptions regarding MAs in Section 2.3: *i*) trains connected to the trackside always move within the assigned MAs, and *ii*) to connected trains, the trackside will assign MAs between the currently known position and a succeeding free VSS or grant an OS MA. Proving these properties required the additional restriction *iii*) that OS MAs are never assigned. This should be expected since this would allow trains to freely move ignoring trackside information. Electrum allows the definition of assertions as regular formulas, which given these pre-conditions can be encoded as:

```
assert trains_Occupied {
  (init and always
      (no mute and no disintegrated and no t:Train | after OS[t])) =>
    always Train.(pos_front+pos_rear).state = Occupied }
check trains_Occupied for 8 VSS, 3 TTD, 2 Train, exactly 12 Time
```

where state predicate `init` forces all trains to be reporting and the track to have a consistent state in the initial state.

More interesting safety properties allow failures in communication or non-integral trains, which necessarily involves reasoning about timers. Recall that our model, in order to be flexible, did not impose any particular duration on the timers, i.e., the number of steps that the starting condition must hold in order to the timer to expire. Since timer duration necessarily affects the correctness of the system, our safety assertions assumed a conservative approach where every timer expires instantaneously (predicate `auto_timer`). Guaranteeing these properties required however the additional pre-condition that *iv*) disconnected trains do not move outside the assigned MA. This allowed us to show that, even with problematic trains, the state is correctly assigned to the VSSs, for instance, that the free state is never assigned to a VSS with a train on it:

```
assert timers_Free {
  (init and always (auto_timer and
    (all t:Train | t.pos_front in MAs[t] and not (after OS[t]))) =>
  always Train.pos_front.state != Free }
```

More complex assertions could test alternative timer durations and reason about possible interleaving issues among different kinds of timers.

5 Discussion

Being an extension to Alloy, it is important to compare the verbosity and readability of Electrum models with those developed in standard Alloy. Thus, a similar encoding of the HL3 concept was developed in Alloy as well[5], which, given the complexity of the case study, enabled us to clearly picture the cons and pros of the two languages. The static structural components of the system are identical in either Electrum or Alloy. Differences arise when modelling the dynamic components, as Alloy requires time, evolution and dynamic properties to be explicitly modelled. This would require the explicit declaration of the signature `Time` and the conversion of all variable signatures and fields to a state idiom [4], where, e.g., field `pos_train` would be declared with type `VSS one → Time`. Then, temporal formulas must explicitly quantify over time instants. For instance the `trains_Occupied` assertion in Alloy would take the shape:

```
assert trains_Occupied {
  (init[first] and all s:Time | no mute[s] and
      no disintegrated[s] and no t:Train | OS[s.next,t])) =>
    all s:Time | Train.((pos_front+pos_rear).s).(state.s) = Occupied }
```

The tradeoff is that Electrum does not allow quantification over time instants. In most cases there is an alternative encoding for such expressions using standard LTL. For instance, in Alloy one can retrieve the last state `s` 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, as in a function that retrieves the VSSs currently known to be occupied by a train:

```
fun knownLoc[s:Time,t:Train] : set VSS {
  let s1 = max[s.*prev&t.report_front] |
  t.pos_front.s1 + t.pos_rear.s1 }
```

Electrum does not allow explicit references to time instants, but the same behaviour was encoded using the **`since`** past operator (Fig. 1). Other expressions are necessarily more verbose in Electrum. For instance, evaluating a field `r` over every instant except `t` can be encoded in Alloy as `r.(Time-t) = a`, while in Electrum it would have to be expanded into a sequence of **`after`** expressions.

The Analyzer allowed for the automatic generation of scenarios and checking of assertions through bounded and unbounded model checking. All analyses were run in a quad-core Intel Core i5-4200U Haswell with 4GB RAM, the bounded relying on MiniSAT 2.2.0 and the unbounded on nuXmv 1.1.1. All the 9 scenarios were generated by bounded model checking procedures, since their exact trace

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

length is known, with performance times ranging from 20s for Scenario 1 to 276s for Scenario 9. The safety properties presented in Section 4 were verified through both bounded and unbounded model checking, since the latter provides additional correctness guarantees but has worse scalability. For instance, property `trains_Occupied`, for 5 `VSS`, 2 `TTD` and 2 `Train` elements is verified by the bounded procedure in 49s and by the unbounded in 1273s. Note that such analysis considers every possible track configuration with that number of sections, 8 for this scope. Previously we proposed an automatic decomposed solving strategy [6] that solves each of these configurations in parallel, which allowed the unbounded performance to be cut down to 73s. A larger scope, with 8 `VSS` and 3 `TTD` elements can be verified in 20m by the bounded procedure, analysing all 42 valid track configurations.

Bounded model checking can sometimes have unpredictable effects for those unaccustomed with its semantics. As already reported, the infinite traces imposed by global constraints forbid deadlocks at the last state, forcing the trace to loopback into a previous state. Since this not necessarily true in every trace, it may lead to unexpected unsatisfiable commands and longer traces. A related issue regards the use of past-time operators which, due to the finite nature of the trace and the alternative past state when reasoning at the loopback state, can also lead to unpredictable behaviour.

6 Conclusions

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 also 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 delays on TTD reports and forcing independent front and rear reports, although we expect them to have a considerable toll on performance.

The definition of the operational scenarios was the most cumbersome task, so we are currently exploring potential extensions to the language to ease that process, 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.

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.

Acknowledgements The authors would like to thank David Chemouil for the support provided during the model checking of the model. This work is financed by the ERDF – *European Regional Development Fund* through the *Operational Programme for Competitiveness and Internationalisation* - COMPETE 2020 and by National Funds through the Portuguese funding agency, FCT - *Fundação para a Ciência e a Tecnologia* within project POCI-01-0145-FEDER-016826.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 117–148 (2003)
2. EEIG ERTMS Users Group: Hybrid ERTMS/ETCS Level 3 – Principles (2017)
3. INESC TEC, ONERA: Electrum Analyzer, v1.0. Available under the MIT License at `https://github.com/haslab/Electrum/releases/tag/v1.0` (2018)
4. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, revised edn. (2012)
5. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: SIGSOFT FSE. pp. 373–383. ACM (2016)
6. Macedo, N., Cunha, A., Pessoa, E.: Exploiting partial knowledge for efficient model analysis. In: ATVA. LNCS, vol. 10482, pp. 344–362. Springer (2017)
7. Moreira, J.M., Cunha, A., Macedo, N.: An ORCID based synchronization framework for a national CRIS ecosystem. F1000Research 4(181) (2015)