Behaviour-Driven Formal Model Development of the ETCS Hybrid Level 3

Michael Butler, Dana Dghaym, Thai Son Hoang, Tope Omitola, Colin Snook
University of Southampton
Southampton, UK
{mjb, d.dghaym, t.s.hoang, t.omitola, cfs}
@soton.ac.uk

Andreas Fellner, Rupert Schlick, Thorsten Tarrach
AIT Austrian Institute of Technology GmbH
Vienna, Austria
firstname.lastname@ait.ac.at

Tomas Fischer, Peter Tummeltshammer
Thales Austria GmbH
Vienna, Austria
firstname.lastname@thalesgroup.com

***Abstract*—Behaviour driven formal model development (BDFMD) enables domain engineers to influence and validate mathematically precise and verified specifications. In previous work we proposed a process where manually authored scenarios are used initially to support the requirements and help the modeller. The same scenarios are used to verify behavioural properties of the model. The model is then mutated to automatically generate scenarios that have a more complete coverage than the manual ones. These automatically generated scenarios are used to animate the model in a final acceptance stage. In this paper, we discuss lessons learned from applying this BDFMD process to a real-life specification: The European Train Control Systems (ETCS) Hybrid Level 3. During the case study, we have developed our understanding of the process, modifying the way we do some stages and developing improved tool support to make the process more efficient. We discuss (1) the need for abstract scenarios during incremental model development and verification, (2) tools and techniques developed to make the running of scenarios more efficient, and (3) improvements to tools that generate new test cases to improve coverage.**

***Index Terms*—Event-B, UML-B, MoMuT, BDFMD, Scenario, ETCS Hybrid Level 3**

I. Introduction

For complex computing systems, formal modelling supports a rigorous system-level engineering method to ensure that the system upholds important properties such as safety and security. Using theorem provers, such properties can be proven to hold generically without instantiation and testing. However, the human centric processes of understanding a natural language or semi-formal requirements document and representing it in mathematical abstraction is subjective and intellectual, leading to misinterpretation. Hence it is vital that domain experts validate the final models to show that they capture the customer requirements.

A widely-used and reliable validation method is acceptance testing, which with adequate coverage, provides assurance that a system, in our case embodied by a formal model, represents the informal customer requirements. Acceptance tests describe a sequence of simulation steps involving concrete data examples to exhibit the functional responses of the system. However, acceptance tests can also be viewed as a collection of scenarios providing a useful and definitive specification of the behavioural requirements of the system. The high level nature of acceptance tests, which are both human-readable and executable, guarantees that they reflect the current state of the product and do not become outdated. They are also necessarily precise and concise to ensure that the acceptance tests are repeatable.

Behaviour-Driven Development (BDD) [14], [17] is a software development process based on writing precise semi-formal scenarios as a behavioural specification and using them as acceptance tests. In [20], we proposed a Behaviour-Driven Formal Model Development (BDFMD) process where manually authored scenarios are used initially to support the requirements and help the modeller and then used to verify behavioural properties of the model. Model mutation is then used to automatically generate further scenarios that have a more complete coverage than the manual ones. The additional scenarios are checked by domain experts to ensure they represent desired behaviour and then used to animate the model in a final acceptance stage. For this acceptance stage, it is important that a domain expert decides whether or not the behaviour is desirable.

In this paper we develop the BDFMD process further based on a 'real-life' case study: The Hybrid ERTMS/ETCS Level 3 (HL3) railway specification. As a result of feedback from the formal model verification using theorem provers, that we reported in [4], the HL3 specification has been re-issued to clarify ambiguities and remove some contentious situations. Here, we focus on validation and how it may guide the verification. We discuss further insights about the process which were revealed by the case study and develop the techniques and tool-support for creating, managing and running scenarios via animation of the models. The contributions of this paper are thus: a realistic case study resulting in; a proposal for abstract scenarios in order to better guide the development and verification of the model at abstract refinement levels; development of the BDFMD process especially in the management and use of scenarios; improved tool support for managing and generating scenarios.

The remainder of the paper is structured as follows. Section II gives an overview of the case study and a summary

of the tools and languages used in the process. Section III describes our BDFMD approach and how it has developed compared to [20]. Section IV introduces our formal model and, in more detail, how we tackled each stage of the BDFMD process on the case study. Section V describes further work that has arisen from the case study and Section VI concludes.

II. Background

In this section, we first give a description of the HL3 case study (Section II-A) and then outline the background technologies that we used in the case study (Section II-B).

A. HL3 Case Study Description

This paper uses the HL3 case study [6], [10] to apply a process pattern for systematic verification and testing. The case study concerns the European Rail Traffic Management System (ERTMS)[1], the system of standards for management and interoperation of signalling for railways by the European Union (EU). HL3 is a "fixed virtual block" approach to train movement, where the Trackside Train Detection (TTD) derived from wayside equipment is augmented by information obtained from trains[2]. Trains equipped with ERTMS equipments can regularly report its current position and integrity status. The scope of the HL3 specification does not include any continuous domain aspects or real-time performance requirements. Timer functionality is described but not quantified. For example, "if the propagation timer expires, X should happen".

The hardware derived TTD section is divided into a fixed number of Virtual Sub-Sections (VSSes). A train movement controller called the Radio Block Centre (RBC) manages the Movement Authority (MA) granted to each train in mission. An Full Supervision Movement Authority (FS MA) is the permission for a train to move safely to a specific location avoiding train collisions when the track is known to be free. An On Sight Movement Authority (OS MA) permits the driver to proceed with caution when the track state is uncertain. However, in order for the RBC to grant a FS MA, it needs to know which sections are free. The status of the VSSes are calculated by the Virtual Block Detector (VBD) depending on the information it receives from the environment:

- Track occupancy received from the trackside.
- Position reports and integrity confirmations received from the trains.
- Timer expiry.

The RBC uses free sections to calculate the FS MA. In addition to the occupied/free VSS states, the VSSes can also be described as unknown or ambiguous. These additional states are necessary to mitigate against possible roll-back of disconnected trains, and to optimise the use of sections in a safe manner.

[1]http://ertms.net.

[2]Trains may or may not be specially equipped with the necessary equipment, hence the term hybrid.

B. Background Technologies

In this sub-section, we give an overview of the background technologies supporting our case study. These include Event-B (Section II-B1) and iUML-B (Section II-B2) modelling methods; ProB model checker (Section II-B3); MoMuT (Section II-B4), a test case generation tool for deriving tests from behavioural models; Cucumber for Event-B/iUML-B (Section II-B5) for the automatic execution of Gherkin scenarios on Event-B/iUML-B models; and BMotion Studio (Section II-B6) for visualising of formal models.

1) Event-B: Event-B [1] is a formal method for system development. An Event-B model contains two parts: *contexts* and *machines*. Contexts contain *carrier sets* s, *constants* c, and *axioms* A(c) that constrain the carrier sets and constants. Note that the model may be underspecified, e.g., the value of the sets and constants can be any value satisfying the axioms. Machines contain *variables* v, *invariants* I(v) that constrain the variables, and *events*. An event comprises a guard denoting its enabling-condition and an action describing how the variables are modified when the event is executed. In general, an event e has the following form, where t are the event parameters, G(t, v) is the guard of the event, and v :=E(t, v) is the action of the event.

any t **where** G(t,v) **then** v :=E(t,v) **end**

Actions in Event-B are, in the most general cases, non-deterministic [9], e.g., of the form v :∈E(v) (v is assigned any element from the set E(v)) or v :|P(v,v') (v is assigned any value satisfying the before-after predicate P(v,v')). A special event called INITIALISATION without parameters and guards is used to put the system into the initial state.

A machine in Event-B corresponds to a transition system where *variables* represent the state and *events* specify the transitions. Event-B uses a mathematical language that is based on set theory and predicate logic.

Contexts can be *extended* by adding new carrier sets, constants, axioms, and theorems. Machines can be *refined* by adding and modifying variables, invariants, events. In this paper, we do not focus on context extension and machine refinement.

Event-B is supported by the Rodin Platform (Rodin) [2], an extensible open source toolkit which includes facilities for modelling, verifying the consistency of models using theorem proving and model checking techniques, and validating models with simulation-based approaches.

2) iUML-B: iUML-B [15], [18], [19] provides a diagrammatic modelling notation for Event-B in the form of state-machines and class diagrams. The diagrammatic models are contained within an Event-B machine and generate or contribute to parts of it. For example a state-machine will automatically generate the Event-B data elements (sets, constants, axioms, variables, and invariants) to implement the states while Event-B events are expected to already exist to represent the transitions. Transitions contribute further guards and actions representing their state change, to the events that they elaborate. A choice of two alternative translation

encodings are supported by the iUML-B tools. State-machines are typically refined by adding nested state-machines to states. Class diagrams provide a way to visually model data relationships.

3) ProB: Consistency of Event-B models is provided via means of proof obligations, e.g., invariant preservation by all events. Proof obligations can be discharged automatically or manually using the theorem provers of Rodin. Another important tool for validation and verification of our model is ProB [13]. ProB provides model checking facility to complement the theorem proving technique for verifying Event-B models. Features of the ProB model checker include finding invariant violations and deadlock for multiple refinement levels simultaneously. Furthermore, ProB also offers an animator enabling users to validate the behaviour of the models by exploring execution traces. The traces can be constructed interactively by manual selection of events or automatically as counter-examples from the model checker. Here, an animation trace is a sequence of event execution with parameters' value. The animator shows the state of the model after each event execution in the trace. Other technologies such as Cucumber for Event-B/iUML-B, BMotion Studio, and our newly developed Scenario checker are built on top of ProB.

4) MoMuT: MoMuT is a test case generation tool able to derive tests from behaviour models. The behaviour model represents a system specification, the generated tests can be used as black box tests on an implementation. They help to ensure that every behaviour that is specified, is also implemented correctly.

In contrast to other model based testing tools, the generated test cases do not target structural coverage of the model, but target exposing artificial faults systematically injected into the model. These faults are representatives of potential faults in the implementation; a test finding them in the model can be assumed to find its direct counterpart as well as similar, not only identical problems in the implementation [7].

As input models, MoMuT accepts Object Oriented Action Systems (OOAS) [11], an object oriented extension of Back's Action systems [3]. The underlying concepts of Action systems and Event-B are both closely related to Dijkstra's guarded command language [5]. For a subset of UML, for some Domain Specific Language s (DSLs) and for a subset of Event-B, transformations into OOAS are available.

MoMuT strives to produce effective tests, i.e. tests exposing faults, as well as efficient tests, i.e. keeping the test suite's size close to the necessary minimum. Thereby, the tests are also suitable as manually reviewed acceptance tests.

5) Cucumber for Event-B/iUML-B: In [20], we described our specialisation of Cucumber for Event-B and iUML-B with the purpose of automatically executing of scenarios for Event-B and iUML-B models. Cucumber [21] is a framework for executing acceptance tests written in Gherkin language and provides Gherkin language parser, test automation as well as report generation. We provide Cucumber step definitions for Event-B and iUML-B in [8] allowing us to execute the Gherkin scenario directly on the Event-B/iUML-B models. Some main Cucumber step definitions are as follows.

- The Cucumber step definitions for Event-B allow to execute an event with some contraints on the parameters, or to check if an event is enabled/disabled in the current state, or to check if the current state satisfies some constraint.
- Cucumber for iUML-B class diagrams provides step definitions for calling a method with some constraints on the method's parameters, or to check the value of an attribute or associations of the class.
- Cucumber for iUML-B state machines provides step definitions for invoking a transition with some constraints on the transition's parameters, or to check the current state of the state-machine.

6) BMotionStudio: In this paper we have used BMotion Studio [12] to create a *domain specific visualisation* (DSV) of our Event-B model. BMotion Studio comes with a graphical environment including a visual editor that provides various *graphical elements* to create a visualisation of the model. A graphical element is based on Scalable Vector Graphics (SVG) and HTML, two markup languages which support widgets like shapes, images, labels, tables and lists. Moreover, *observers* are used to link the model with the visualisation. For instance, the tool provides a *formula observer* that binds a formula (e.g. an expression or a variable) to a graphical element and allows the tool to compute a visualisation for any given state by changing the properties of the graphical element (e.g. the colour or position) according to the evaluation of the formula in the respective state. Finally, *event handlers* can be attached to the visualisation to provide interactive functionalities, such as an *execute event handler* that binds an Event-B event to a graphical element and executes the event when the user clicks on the graphical element.

III. A Process Pattern for Systematic Verification and Testing

In our previous work [20], we presented an approach for formal systems modelling and validation based on BDD. In this paper, we extend the forementioned approach and defined a process pattern for systematic verification and testing (Fig. 1). Note that the pattern is generic in terms of the methods or tools used in different steps. The steps of the process are as follows.

1) In the *scenario modelling* step, the *manually written scenarios* are produced from the *system requirements*. The output manual scenarios are kept close to the terminology and representation of the domain and act as a good means to support communication between domain experts and development experts. In our HL3 case study, the scenarios are given in [6]. However, in hindsight, the descriptions of the scenarios do not always use a consistent terminology which, for a non-domain expert, makes them difficult to follow and introduces ambiguities. It would be better to rewrite the scenarios

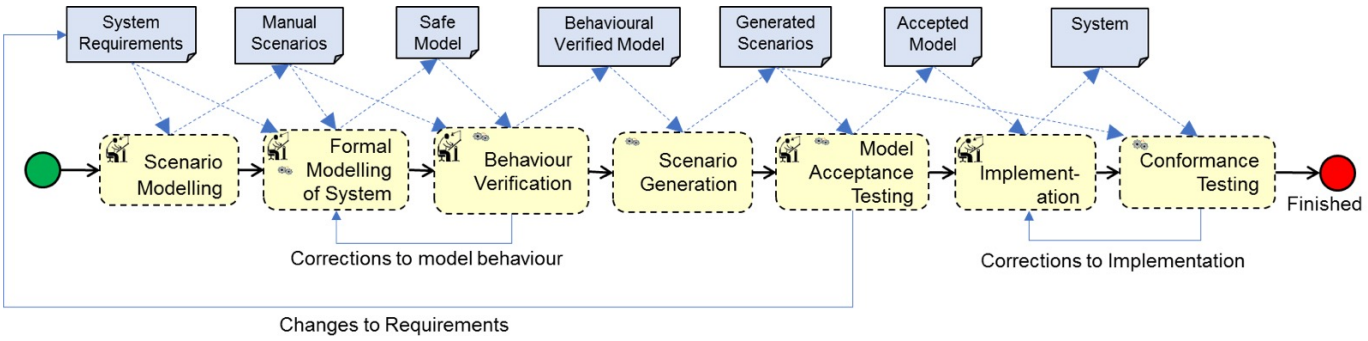

Fig. 1. A Process Pattern for Systematic Verification and Testing

in a more concise and precise language, e.g., a Gherkin domain-specific syntax for European Train Control System (ETCS) systems.

2) In the *formal modelling* step, the model is produced from *system requirements* and the manual scenarios. The output of the modelling step is a *safe model*, in the sense that it is fully proven to be consistent with the invariants. We use "safe" in a wide sense to include any important properties). In our case study, we use Event-B/iUML-B (Sections II-B1 and Section II-B2) as our modelling methods.
3) The safe model is *behaviourally verifed* against the manually written scenarios. The purpose is to verify that the safe model exhibits the behaviour specfied in the requirements which cannot be expressed via invariants. The output of this step is a (safe and) *behaviourally verified model*. In our case study, we use Cucumber for Event-B/iUML-B for *automatically verifying* the behaviour of our model written in Event-B/iUML-B. At the same time, the newly developed scenario checker (see Section IV-B2) are used to *interactively verify* the behaviour of our model with the assistance BMotion Studio (see Section II-B6). More discussion about our BMotion Studio visualisation in BMotion Studio is in Section IV-B1.
4) The behaviourally-verified model is used as the input for a *scenario generator*, which automatically produces a collection of *generated scenarios*. In our case study, we use an Event-B-enabled version of MoMuT (see Section II-B4) as the scenario generator. The generated scenarios should be reviewed to ensure that they represent desired behaviour. If the model still contains undesirable behaviour, that was not detected in the previous step, this will be reflected in the generated scenarios. Note that the set of generated scenarios is a super-set of the manually scenarios. This is for the purpose of using the "complete" set of scenarios in the later steps such as model acceptance testing and conformance testing.
5) The generated scenarios are used for *acceptance testing* of the behaviourally verified model. Model acceptance testing allows stakeholders to assess the usefulness of the model by watching its behaviour. We again use Cucumber for Event-B/iUMLB to automatically illustrate the generated scenarios to different stakeholders. The scenarios are in "natual language" and it is easy to see the correspondence between the scenarios and the requirements. The output of this step is an *accepted model*, in the sense that it has been approved by the stakeholders after validating its behaviours using the generated scenarios.
6) In the *implementation* step, the accepted model is used as the input to produce the *system*. In our case study, while there is no implementation yet, the accepted model can be used, for example, in conjunction with a code generator to produce an implementation. Note that the accepted model often contains environment and the controller. In this case, we will need to separate the controller model and implement it.
7) Finally, the system is *conformance tested* against the scenarios. This is to ensure that the implementation is consistent with the accepted model with respect to the scenarios. Any inconsistency found in this step will require corrections to the implementation and we expect iterations between the implementation and conformance testing steps. Note that the conformance test is for the implementation of the controller against the environment model with respect to the scenarios. As a result we will need to co-simulate the environment together with the controller implementation for testing the their conformance. Implementing the HL3 controller and testing its conformance is our future work (see Section V).

Compared to [20], our process pattern here extends the original approach by having additional steps for scenario modelling, implementation, and conformance testing. We also explicitly added the iterative nature between the different steps, e.g., between formal modelling and behavioural verification, between implementation and conformance testing. This is an indication that the process is not a one-directional procedure: iterations are required when modifications need to be made to the different artefacts during development.

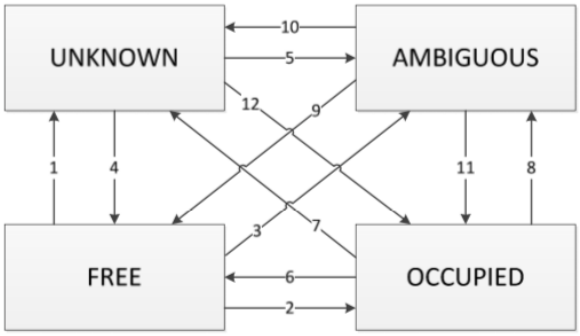

Fig. 2. VSS State Diagram from HL3 specification [6]

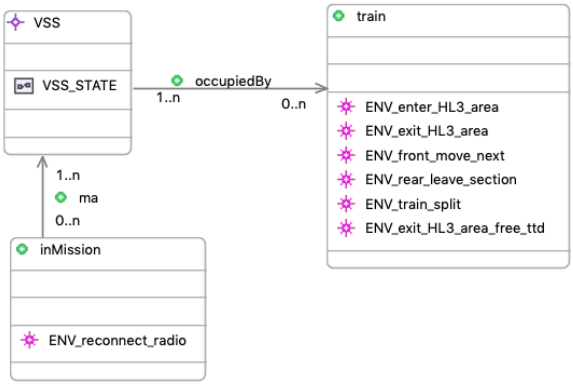

Fig. 3. iUML-B class diagram showing the VSS class

IV. Case Study

We formally model the HL3 case study in [4]. Our model follows a refinement-based approach using Event-B, where the main focus is the VBD and how it can predict the VSS state based on the provided information.

First, we start by modelling the environment focusing on train movement, the actual position of the trains relative to the VSS sections and TTDs. We also introduce radio communication to distinguish between connected and disconnected trains which can only be detected by the trackside equipement. Later, we introduce the concept of FS MA to control the movement of trains and OS MA to permit the driver to control the movement of trains when the track state is uncertain. Finally, we introduce the VBD, which uses the information it receives from the environment, in the form of position and integrity reports, trackside vacancy and timers to calculate the VSS states. The VSS state is determined by a fully connected state machine (Fig. 2) described in [6]. The transition from one state to the other can only happen under certain conditions, which are modelled as event guards in Event-B. The VSS states are eventually fed to the RBC to grant movement authorities. Fig. 3 illustrates a part of our iUML-B model for the VSS.

A. Modelling with Scenarios

While constructing the formal model, we were only able to use the scenarios to a limited extent. They were useful to help understanding of the specification but, because they are very concrete, we could not use them to drive or verify our abstract models. Only when we had reached a refinement level that implemented the internal VBD processing of VSS state, could we animate the scenarios in our models. We did not use the scenarios to help find useful abstractions during modelling, for this we relied on analysing the main text in the specification. However, it may be useful to find a way to abstract from the given scenarios to form more abstract ones.

Our refinement strategy introduces new details with each refinement level, in addition to some data refinement. Each level focuses on a newly introduced concept. To illustrate our abstract scenarios, we take Steps 2 and 3 from Scenario 2 of [6], which are described concretely as follows. An illustration of the first three steps of Scenario 2 is in Fig. 4. (Note that the bracketed numbers in the text refer to subclauses of transitions in the state diagram of Fig. 2. For example, (#8A) refers to transition 8 firing because a particular disjunct, A, of its guard has become true.

1) **Step 2**: Train 1 and Train 2 are split. Train 1 remains connected with trackside and reports the new train data train length. Except of the reporting of the mode change, Train 2 is not connected to the trackside. Due to the reported change of train data train length VSS 12 becomes “ambiguous” (#8A). The change of train data train length also starts the integrity loss propagation timer for VSS 12.
2) **Step 3**: Train 1 receives an FS MA until end of VSS 33, starts to run again, passes the TTD section border, and reports its position on VSS 21, which becomes “ambiguous” (#3A). VSS 12 becomes “unknown” (#10A).

Following our refinement strategy these concrete descriptions of the steps can be abstracted as follows, where each level adds more detail (shown in italics) corresponding to the functionality added in that refinement level of the model.

1) Movement & VSS:
   - *Train 1 and 2 are split.*
   - *Train 1 starts to run again and moves to VSS 21.*
2) Radio Communication
   - Train 1 and 2 are split. *Train 1 remains connected with the trackside. Train 2 is not connected to the trackside.*
   - Train 1 starts to run again and moves to VSS 21.
3) TTD
   - Train 1 and 2 are split. Train 1 remains connected with the trackside. Train 2 is not connected to the trackside.
   - Train 1 starts to run again, *passes the TTD section border* and moves to VSS 21.
4) Mission & Movement Authority
   - Train 1 and 2 are split. Train 1 remains connected with the trackside. Train 2 is not connected to the trackside.

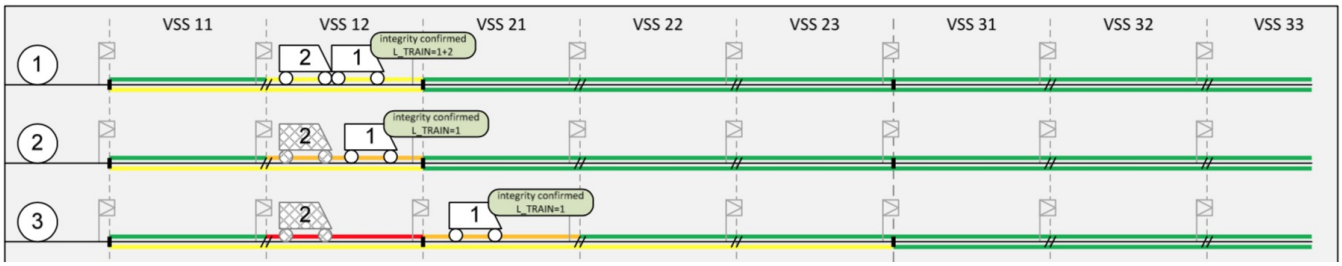

Fig. 4. Scenario 2 - Splitting of a composite train with integrity confirmed by external device [6] (First three steps)

- Train 1 *receives an MA until end of VSS 33*, starts to run again, passes the TTD section border and moves to VSS 21.

5) Position Reports
   - Train 1 and 2 are split. Train 1 remains connected with the trackside and reports its position in VSS 12. Train 2 is not connected to the trackside.
   - Train 1 receives an MA until end of VSS 33, starts to run again, passes the TTD section border and *reports its position on VSS 21*.
6) VSS Availability & MA types
   - Train 1 and 2 are split. Train 1 remains connected with the trackside and reports its position in VSS 12. Train 2 is not connected to the trackside. *VSS 12 remains unavailable.*
   - Train 1 receives an FS MA until end of VSS 33, starts to run again, passes the TTD section border and reports its position on VSS 21, *which becomes unavailable. VSS 12 remains unavailable*.
7) Integrity
   - Train 1 and 2 are split. Train 1 remains connected with the trackside and reports the new train data length. Train 2 is not connected to the trackside. VSS 12 remains unavailable.
   - Train 1 receives an FS MA until end of VSS 33, starts to run again, passes the TTD section border and reports its position on VSS 21 *with integrity confirmed*. VSS 21 becomes unavailable. VSS 12 remains unavailable.
8) Timers
   - Train 1 and 2 are split. Train 1 remains connected with the trackside and reports the new train data length. Train 2 is not connected to the trackside. VSS 12 remains unavailable. *The change of train data length also starts the integrity loss propagation timer for VSS 12.*
   - Train 1 receives an FS MA until end of VSS 33, starts to run again, passes the TTD section border and reports its position on VSS 21 with integrity confirmed. VSS 21 becomes unavailable. VSS 12 remains unavailable.

Using these abstract versions of the scenarios, we could animate our model and incrementally verify that its behaviour is developing in accordance with the scenario. Finally, at the last refinement we verify the model against the complete concrete scenario described in the specification document.

However, this approach is retrospective; we abstracted the scenario to match our refinement strategy. It may also be possible to abstract the scenarios first, for example, by focusing on different areas of functionality in each scenario and then looking for common patterns across the set of scenarios. The abstract scenarios could then be used as suggested in Section III, step 2, to guide the refinement strategy used in the model.

B. Behaviour Verification and Acceptance Testing

This section describes our approach, and tool-support for, behaviour verification and acceptance testing. These steps both rely on being able to animate the model in controlled steps that make up a particular scenario. As the scenario unfolds, at each step the state of the model is examined to ensure that it is a valid representation of the desired system. The scenario must also be feasible for it to complete. That is, the steps of the scenario must be enabled in the model at the appropriate stages. This examination of state and enabledness, may either be performed by manual observation (initial behavioural verification and acceptance tests) or by comparison with previous animations (for regression testing of the model after changes). Manual observation is assisted by a BMotion Studio visualisation of the state as described in Section IV-B1, whereas comparison is done automatically by running the cucumber execution tool described in Section II-B5 or (more slowly) by replaying the scenario using the scenario checker tool described in Section IV-B2. The scenarios may either be written 'by hand' by authoring a cucumber for Event-B script, or recorded by stepping through the animation using the scenario checker tool described in Section IV-B2 and saving the results.

1) Visualisation: In order to efficiently verify or validate the behaviour of the model we need to be able to visualise its state while running scenarios. The ProB animator provides a 'state view' window (lower right view pane in Fig. 5) showing the current and past state of each variable. However, this view shows the mathematical forms such as sets of pair maplets for a relation, which is difficult to interpret with respect to the domain being modelled. To visualise the state in a more appealing way we created a BMotion Studio visualisation

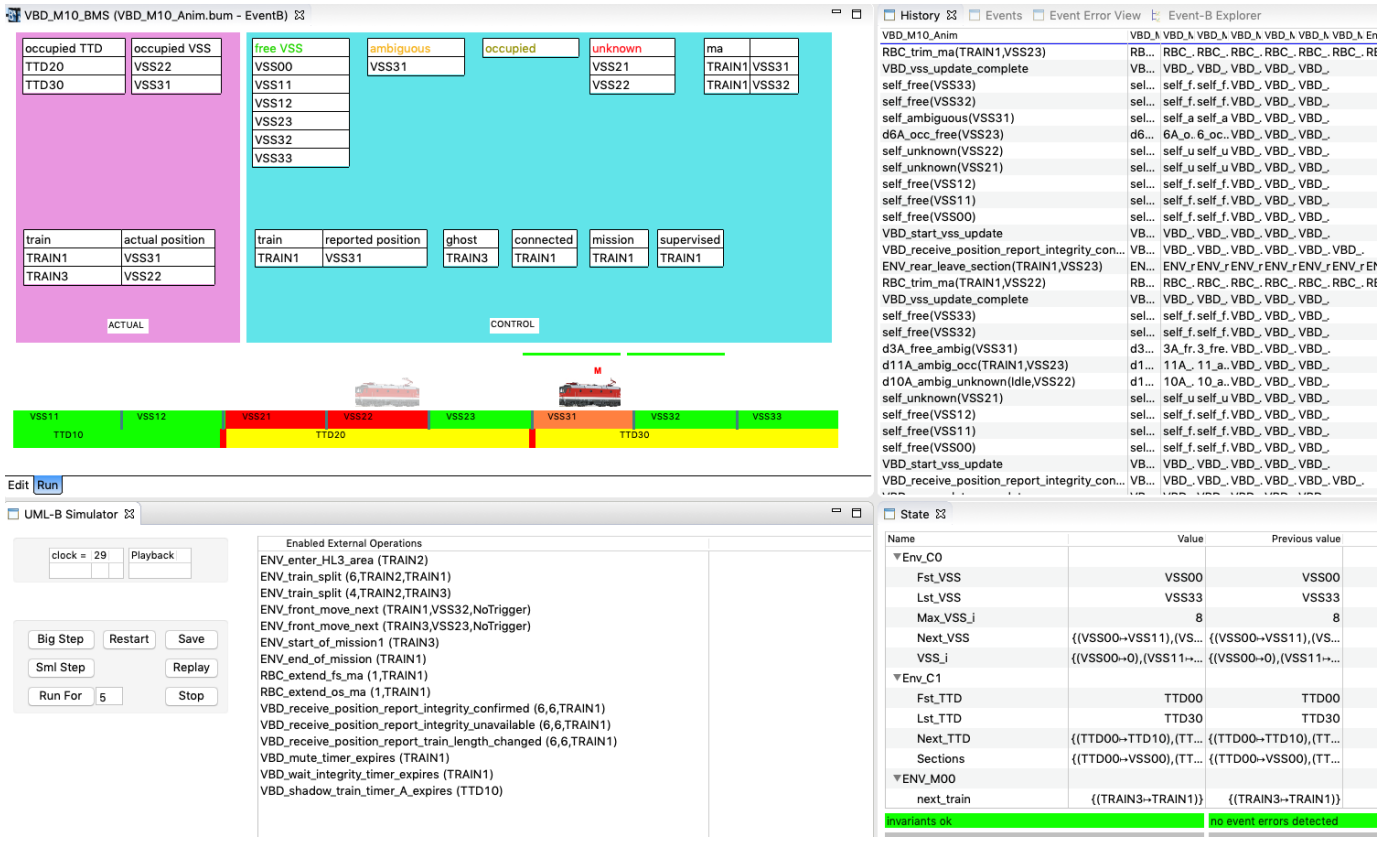

Fig. 5. Behaviour Verification using BMotion Studio Visualisation and Scenario Checker to drive ProB

(upper left view pane in Fig. 5) that is linked to the variables of the model.

For the incremental behaviour verification stage, while constructing the model, we used a simple tabular representation of the data which is very quick to create but easier to map to domain concepts than lists of maplet pairs. New tables could be added easily to represent the new variables introduced at each refinement level. We arranged the tables into coloured blocks to easily distinguish the representation of the physical trains in the environment from the representation of the internal state of the control system. This stage is mainly for the modeller who has a good understanding of the variables in the model.

For the acceptance testing stage, where stakeholders such as domain experts are expected to understand the behaviour of the model, a more concrete visualisation of state was added consisting of coloured blocks to represent the state of TTD and VSS, coloured line to indicate movement authority, and a moving picture of a train to represent train position on the sections. The trains also change their rendering to indicate whether they are 'in mission' or not and whether they are 'ghost'. The visualisation was used to demonstrate and successfully validate the model to domain experts from equipment suppliers and national rail operators.

2) Scenario Checker: The BDFMD process relies heavily on animation of scenarios. Hence it is important to make this activity as efficient as possible. ProB provides a basic GUI interface for controlling animations, but manually firing the correct sequence of selected events was found to be a slow and onerous task. ProB also provides an API for extending the tooling with additional facilities and this was used to provide a new 'Scenario Checker' tool (lower left view pane in Fig. 5). The tool provides two areas of functionality that make scenario checking feasible: run to completion and record/replay.

In Event-B models we model closed systems of interacting components including the domain or environment and any controlling device. Event-B makes no distinction between the kinds of events and hence an event in the environment will usually trigger a sequence of events of the controller as it responds to the change in the environment. The events that fire to implement control are considered internal implementation steps and are not specified in our scenarios. The scenario checker repeatedly automatically fires any internal events until none are enabled and then waits for another external environment event to be selected. (For example, in Fig. 5, compare the enabled external events available in the scenario checker with the full history of fired events shown by ProB in the upper right view pane). Only the external environment events and associated state are recorded during a scenario. During playback, only the external events are needed from the scenario script because internal ones are fired automatically when enabled.

In our Event-B model we left the order of firing of some internal process steps non-deterministic because it is not important for the refinement proof and it is cumbersome to specify the negation of all other options. In the scenario checker tool we added a priority annotation mechanism to ensure that the correct order is used for animation.

The scenario checker saves a snapshot of the state at each step that is recorded. This can then be used during playback to compare the new state of the running model with that recorded previously. Variables in the model either represent physical properties of the environment or internal processing of the controller. The scenario checker includes the option to annotate some variables as *PRIVATE* so that they are not recorded or checked. Any variable may be excluded in this way but typically the environment variables need to be checked and internal process control variables (akin to programme counters) should be ignored. Control state variables that represent the controllers model of the environment may be ignored or checked depending on whether they are significant to the specification being modelled. In our case we ignored the process variables that control the sequence of VSS state updates but checked the controller state variables such as VSS state, ghost train, in mission etc. since these are properties that the controller must understand according to the HL3 specification.

For now, we have used comments to annotate which are the internal events and which are the private variables. The annotations are made in a special refinement which also instantiates the model with the example track sections used in the scenarios. In future we intend to add a modelling feature to do this in a more robust way using an attribute of the meta-class. Another model feature that would be useful to 'hide' during scenario checking is local variables. Event-B syntax does not distinguish between parameters that represent choices in the environment and those that are used for convenience as local variables. It would be useful to only record the former and hide the latter when scenario checking. However, due to event extension, parameters can not be annotated in the final animation refinement since they are not repeated at subsequent refinements of the event.

The current implementation records the scenario scripts in an Eclipse Modelling Framework (EMF) based format. In future work we will use the Cucumber for Event-B notation as the persistence of the scenario checker tool to improve the integration of these tools. For example, the scenario recording mechanism could then be used to create cucumber scenarios that are used like regression tests automatically replayed after model changes in a continuous integration environment.

C. Scenario Generation

This section describes our experience of using the MoMuT test-case generator to generate scenarios for the HL3 case study.

1) Partial Order Model Adaption: Requirements can be underspecified with regards to the order of events. For example, two lights, A and B, could be required to be switched on in an arbitrary order before some next event C happens. Both sequences A-B-C and B-A-C are correct implementations of this requirement, yielding the same final result. The number of correct sequences grows exponentially with the number of events that are allowed to appear in an arbitrary order, which makes model exploration highly inefficient. If the events are independent (i.e. one event does not modify a variable that the other event reads or modifies), MoMuT is able to identify such sequences using partial order reduction, However, the transition events for updating the VSS states are not independent because they all update a single set variable to indicate completion. In order to make model exploration more efficient, in our experiments, instead of using one separate flag per event, we refined the model to strictly order the events according to the index of the VSS. Since this is a valid refinement that reduces non-determinism, safety properties are not affected. Possible implementations are reduced since any implementation should obey this arbitrarily chosen ordering. However, since these events are annotated as internal, their order is not checked during conformance testing using these scenarios.

2) Model Transformation: The native modeling formalism of MoMuT is object oriented action systems (OOAS). Therefore, in order to apply test case generation, an automatic transformation from Event-B to OOAS was implemented. While OOAS and Event-B are similar, the modeling formalism have a few differences in their workings and semantics. Firstly, in Event-B, types of event parameters are implicitly given by constraints like set membership, whereas in OOAS, types, including list sizes, need to be explicitly given. Secondly, in Event-B, the right hand side of assignments always refers to the value before activating the event, whereas in OOAS, the right hand side of an assignment immediately updates the value of variables. Therefore, ordering of assignments plays a role. Thirdly, in Event-B, sets are heavily used, whereas in OOAS sets are not available as a primitive data type. Instead, the transformation emulates sets with lists. MoMuT explores models in an explicit state and enumerative manner. This makes constructs like nested set comprehensions – which are heavily used in Event-B– especially for larger sets or numeric value ranges, very inefficient. To address this issue, we manually reformulated these constructs to use nested fold operations without set comprehension.

3) Scenario recording: In lieu of a finalised transformation from Gherkin or from the persistence format of the scenario checker to the test format of MoMuT, the tests were recorded using MoMuT's text interface based animation feature.

4) Coverage Analysis and Incremental Test Case Generation: In a first round of experiments, the test suite was extended by four scenarios, improving coverage and strong kills. Analysis of not covered model parts exposed a mistake in the model that made part of the model unreachable. Fixing this led to a changed model with a slightly higher number of mutants (before: 2366, now 2370).

In the second round of experiments, mutation coverage of the manual scenarios has been evaluated again; 3 manual

Scen	Strong	Fail	Weak	Equiv	Missed	Steps
m1	16.03	1.1	4.05	17.85	60.97	253
m2	7.43	0.89	6.37	16.2	69.11	244
m3	8.69	1.31	1.14	21.9	66.96	311
m4	7.22	0.97	2.91	11.22	77.68	165
m5	1.01	0.68	4.35	14.73	79.24	219
m6	5.91	0.93	3.76	13.92	75.49	206
m7	1.39	0.72	2.45	13.38	82.07	209
m8	1.6	0.34	1.77	11.31	84.98	474
m9	1.43	0.68	3.16	12.28	82.45	363
man.	50.72	2.03	8.99	29.66	8.61	2444
gen.	30.3	0.0	0.93	2.66	66.12	15416
all	57.97	2.07	11.9	27.09	0.97	17860

TABLE I
COVERAGE AND SIZE DATA OF MANUAL AND GENERATED SCENARIOS

scenarios have been added, and 75 additional scenarios have been generated automatically to increase the coverage. Table I summarizes coverage for the original manual scenarios (m1 ... m9), combined original manual scenarios (man.), combined new manual and generated scenarios (gen.), and the combination of all scenarios (all).

For each row, the following values are given in percentages of mutants with respect to the total number of mutants (2370): Strong kills (**Strong**) shows how many mutants changed behaviour on the outputs. Failures (**Fail**) shows how many mutants were defective and produce e.g. overflows. Weak kills (**Weak**) show how many mutants change the inner status of the model, but do not propagate this change to the outputs. Locally equivalent (**Equiv**) shows how many mutants do not diverge from the original model in inner state for the given scenarios. **Missed** shows how many mutants were not visited at all by the scenario. Finally, **Steps** shows the (combined) length of the test (test suite).

The final set of scenarios significantly improves the test suite by reducing the missed mutants from 8.61% down to 0.97% and increasing strong kill coverage up from 50.72% to 57.97%.

The manual scenarios do not visit the mutants of four events at all. The generated scenarios now visit all of these events. Similarly, the initial manual scenarios do not kill any mutants of 12 events. The generated scenarios kill mutants of six of these events.

The test case generation takes several hours on reasonably strong servers. Additional scenario generation campaigns with varied exploration parameters could produce further improved results, pushing some mutants from being weak kills or locally equivalent to strong kills, but will also increase generation time substantially. Several of the locally equivalent mutants might be globally equivalent and cannot be killed by a scenario at all.

5) Validation of model transformation: Both the manual changes to the OOAS model and the manual recording of the scenarios are a potential source of mistakes. In order to catch problems introduced in these steps, the manual and generated scenarios were automatically run against an animation of the Event-B model using ProB's API. This validation step was crucial in model and model transformation development, since it revealed bugs that could then be fixed. All manual scenarios worked correctly for the final version of both models. However, the generated scenarios exposed another difference between the OOAS model and the Event-B model. This shows the value of this validation step, as well as the automatic generation of additional scenarios, in practice.

6) Validation of generated scenarios: The effort for manual validation of the generated scenarios is reasonable. While reviewing 75 scenarios might seem a lot, the overall number of steps in the generated scenarios is around six times the number of steps in the manual scenarios. Since the generated scenarios share large prefixes with manual scenarios and among each other, making this visible to the reviewer of the test would further ease the review process. As the scenarios are needed to achieve the wanted coverage of the model, the alternative to review would be writing those scenarios, which is clearly more effort.

V. FUTURE WORK

Future work will focus on the two contrasting areas discussed in this paper.

1) Research into abstract scenarios will include looking at how they can be synthesised and whether they can be used to guide the modelling process. While it is clear that concrete scenarios help to explain the detailed requirements, it is less clear how abstract scenarios can be derived and used to direct the modelling. Nevertheless, our abstraction based modelling process would benefit from more abstract explanations of the reasons for specification details. This work requires experimental research.

2) Tool and technology improvements are needed to better support the management and use of scenarios for verification and validation. The use of scenarios was clearly beneficial in this case study but limited by the need for more integrated tool support. The three tools (Scenario Checker, Cucumber for Event-B, MoMuT) currently use different syntax for persisting scenarios. It would clearly be of benefit to share a common serialisation (or provide bi-directional transformation) to increase the efficiency of sharing scenarios between tools.

Further work is needed to improve MoMuT's support for Event-B language elements such as relations and to optimize the generator for typical structures produced by iUML-B. We will also investigate the characteristics of mutation operators for Event-B and compare with classic coverage measures such as MC/DC.

Our case study has not yet reached the implementation stage (Section III) steps 6 and 7). We plan to illustrate these steps by decomposing the model into subsystems representing a) the VBD to be implemented and b) the environment that it operates in (including train movement and RBC). We could then 'implement' the VBD subsystem by modelling it in a Modellica modelling tool from which an FMI can be exported. The scenarios can then be used for conformance testing using the co-simulation tools developed by Savicks et al. [16] to animate both the FMI encapsulating the implementation and the environment model still in Event-B.

VI. Conclusion

A behaviour driven approach to formal modelling relies heavily on animating scenarios and the case study has highlighted this, driving us to develop better tools for running and managing scenario scripts. The approach has clear benefits in verifying the model behaviour when the model is first constructed and subsequently in validation by stakeholders. During behaviour verification a number of anomalies were discovered by running the scenarios provided in the specification. For example, scenario 8 highlighted a misunderstanding in our model where the update of VSS in response to reporting of position must be done separately from responses to TTD triggers. An example of a discrepancy that was not covered by the provided scenarios is when a propagation timer changes the state of a VSS that is already allocated in a train's FS MA. In other cases, areas of concern in the specification were identified while attempting to prove safety invariants about FS MA and these were addressed in a new version of the European Train Control System (ETCS) specification illustrating the importance of formal verification by proof. We were able to demonstrate the model to interested parties including railway experts and operators at an open event organised by the Enable-S3 project in Graz, Austria, as well as at the RSSRail 2019 conference in Lille, France.

The case study highlighted a mismatch with the Event-B abstraction approach when only detailed concrete scenarios are available. We were not able to test the full scenarios until the model had been fully developed to that level. To address this, we propose a technique of abstracting from the concrete scenarios to obtain more abstract scenarios that are appropriate for particular refinement levels. This will allow domain experts to engage in the modelling process at an early stage and guide its development. Full automation of replaying scenarios and checking that the expected state is obtained, would allow them to be used in a 'regression testing' mode as used in continuous integration. Further research is needed to discover whether abstract scenarios can be used to *drive* the abstractions or whether it is better to let the model *determine* the abstraction of scenarios.

Acknowledgements

This work has been conducted within the ENABLE-S3 project that has received funding from the ECSEL Joint Undertaking under Grant Agreement no. 692455. This Joint Undertaking receives support from the European Union's HORIZON 2020 research and innovation programme and Austria, Denmark, Germany, Finland, Czech Republic, Italy, Spain, Portugal, Poland, Ireland, Belgium, France, Netherlands, United Kingdom, Slovakia, Norway.

ENABLE-S3 is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program 'ICT of the Future' between May 2016 and April 2019. More information is at https://iktderzukunft.at/en/.

References

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, November 2010.

[3] RJR Back and Kaisa Sere. Stepwise refinement of action systems. In *International Conference on Mathematics of Program Construction*, pages 115–138. Springer, 1989.

[4] Dana Dghaym, Michael Poppleton, and Colin Snook. Diagram-led formal modelling using iUMLB for Hybrid ERTMS Level 3. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z: ABZ 2018*, volume 10817, pages 338–352. Springer, May 2018.

[5] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[6] EEIG ERTMS Users Group. *Principles: Hybrid ERTMS/ETCS Level 3. Ref. 16E042 Version 1A.*, July 2017. http://www.ertms.be/sites/default/files/2018-03/16E0421A_HL3.pdf.

[7] Andreas Fellner, Willibald Krenn, Rupert Schlick, Thorsten Tarrach, and Georg Weissenbacher. Model-based, mutation-driven test case generation via heuristic-guided branching search. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 56–66. ACM, 2017.

[8] Tomas Fischer. Cucumber for Event-B and iUML-B. https://github.com/tofische/cucumber-event-b, 2018.

[9] Thai Son Hoang. An introduction to the Event-B modelling method. In *Industrial Deployment of System Engineering Methods*, pages 211–236. Springer-Verlag, 2013.

[10] T.S. Hoang, M. Butler, and K. Reichl. The hybrid ERTMS/ETCS level 3 case study. In M. Butler, A. Raschke, T.S. Hoang, and K. Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 251–261. Springer International Publishing, 2018.

[11] Willibald Krenn, Rupert Schlick, and Bernhard K Aichernig. Mapping UML to labeled transition systems for test-case generation. In *Formal Methods for Components and Objects*, pages 186–207. Springer, 2010.

[12] Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. Visualising Event-B models with B-Motion Studio. In *Proceedings of FMICS 2009*, volume 5825 of *Lecture Notes in Computer Science*, pages 202–204. Springer, 2009.

[13] Michael Leuschel and Michael Butler. ProB: An automated analysis toolset for the B method. *Software Tools for Technology Transfer (STTT)*, 10(2):185–203, 2008.

[14] Dan North. Introducing BDD. *Better Software Magazine*, March 2006.

[15] Mar Yah Said, Michael Butler, and Colin Snook. A method of refinement in UML-B. *Softw. Syst. Model.*, 14(4):1557–1580, October 2015.

[16] Vitaly Savicks, Michael Butler, and John Colley. Co-simulating Event-B and continuous models via FMI. In *Proceedings of the 2014 Summer Simulation Multiconference*, SummerSim '14, pages 37:1–37:8, San Diego, CA, USA, 2014. Society for Computer Simulation International.

[17] J. F. Smart. *BDD in Action: Behavior-Driven Development for the Whole Software Life cycle*. Manning Publications Company, 2014.

[18] Colin Snook. iUML-B statemachines. In *Proceedings of the Rodin Workshop 2014*, pages 29–30, Toulouse, France, 2014. http://eprints.soton.ac.uk/365301/.

[19] Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, January 2006.

[20] Colin F. Snook, Thai Son Hoang, Dana Dghaym, Michael J. Butler, Tomas Fischer, Rupert Schlick, and Keming Wang. Behaviour-driven formal model development. In Jing Sun and Meng Sun, editors, *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings*, volume 11232 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2018.

[21] Matt Wynne and Aslak Hellesøy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Programmers, LLC, 2012.