Domain-Specific Scenarios for Refinement-based Methods

Colin Snook✉https://orcid.org/0000-0002-0210-0983, Thai Son Hoanghttps://orcid.org/0000-0003-4095-0732, Dana Dghaymhttps://orcid.org/0000-0002-2196-2749, and Michael Butlerhttps://orcid.org/0000-0003-4642-5373

ECS, University of Southampton, Southampton, U.K.,
{cfs, t.s.hoang, d.dghaym, mjb}@ecs.soton.ac.uk

Abstract. Formal methods use abstraction and rigorously verified refinement to manage the design of complex systems, ensuring that they satisfy important invariant properties. However, formal verification is not sufficient: models must also be tested to ensure that they behave according to the informal requirements and validated by domain experts who may not be expert in formal modelling. This can be satisfied by scenarios that complement the requirements specification. The model can be animated to check that the scenario is feasible in the model and that the model reaches states expected in the scenario. However, there are two problems with this approach. 1) The provided scenarios are at the most concrete level corresponding to the full requirements and cannot be used until all the refinements have been completed in the model. 2) The natural language used to describe the scenarios is often verbose, ambiguous and therefore difficult to understand; especially if the modeller is not a domain expert. In this paper we propose a method of abstracting scenarios from concrete ones so that they can be used to test early refinements of the model. We also show by example how a precise and concise domain specific language can be used for writing these abstract scenarios in a style that can be easily understood by the domain expert (for validation purposes) as well as the modeller (for behavioural verification). We base our approach on the Cucumber framework for scenarios and the Event-B modelling language and tool set. We illustrate the proposed methods on the ERTMS/ETCS Hybrid Level 3 specification for railway controls.

keywords: Event-B; Cucumber; Validation; Domain Specific Language

1 Introduction

Abstraction and refinement play a vital role in analysing the complexity of critical systems via formal modelling. Abstraction allows key properties to be established which are then proven to be maintained as system details are gradually introduced in a series of refinements. However, domain requirements are often written in natural language [3] which can be verbose and ambiguous leading

to potential misinterpretation by formal modelling engineers. Hence, model verification is insufficient; validation of the model by domain experts is equally important to ensure that it is a true representation of the system in mind. In previous work [9] we proposed a behaviour driven approach to formal modelling that allows domain experts to drive the formal modelling using scenarios. The model is animated to check that the scenario is feasible and reaches the states expected in the scenario. In this paper we propose the use of a Domain Specific Language (DSL) that can be understood both by domain expert and model engineer and is precise enough to provide a repeatable validation/acceptance test of the formal systems model. Furthermore, we propose a technique of synthesising abstract scenarios from more concrete ones, so that the abstract refinements of the model can be checked at an intermediate stage rather than waiting until the final details have been incorporated. We illustrate the approach using the European Rail Traffic Management System (ERTMS)/European Train Control System (ETCS), Hybrid Level 3 (HL3) specification [7] for which we have previously developed a formal model presented in [4].

The paper is structured as follows: Section 2 provides background on the Event-B formal modelling language, Cucumber framework for scenarios and the HL3 case study. Section 3 introduces the example scenario (from [7]) that we use for illustrating our proposed method. Section 4 illustrates a possible DSL for scenarios of the HL3 model. Section 5 shows how we would describe the example scenario in our DSL. Section 6 presents abstract versions of the concrete scenario to illustrate how these can be systematically deduced to match the refinements in the model. Section 8 describes future work and Section 9 concludes.

2 Background

2.1 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.

$$\textbf{any}\ t\ \textbf{where}\ G(t,v)\ \textbf{then}\ v := E(t,v)\ \textbf{end}$$

Actions in Event-B are, in the most general cases, non-deterministic [8], e.g., of the form $v :\in E(v)$ (v is assigned any element from the set E(v)) or $v :\mid P(v,v')$ (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.

2.2 Cucumber for Event-B

The Behaviour-Driven Development (BDD) principle aims for pure domain oriented feature description without any technical knowledge. In particular, BDD aims for understandable tests which can be executed on the specifications of a system. BDD is important for communication between the business stakeholders and the software developers. Gherkin/Cucumber [10] is one of the various frameworks supporting BDD.

Gherkin [10, Chapter 3] is a language that defines lightweight structures for describing the expected behaviour in a plain text, readable by both stakeholders and developers, which is still automatically executable.

Each Gherkin scenario consists of steps starting with one of the keywords: `Given`, `When`, `Then`, `And` or `But`.

- Keyword `Given` is used for writing test preconditions that describe how to put the system under test in a known state. This should happen without any user interaction. It is good practice to check whether the system reached the specified state.
- Keyword `When` is used to describe the tested interaction including the provided input. This is the stimulus triggering the execution.
- Keyword `Then` is used to test postconditions that describe the expected output. Only the observable outcome should be compared, not the internal system state. The test fails if the real observation differs from the expected results.
- Keywords `And` and `But` can be used for additional test constructs.

In [9], we described our specialisation of Cucumber for Event-B with the purpose of automatically executing of scenarios for Event-B models. Cucumber [10] 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 in [5] allowing us to execute the Gherkin scenarios directly on the Event-B models. 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.

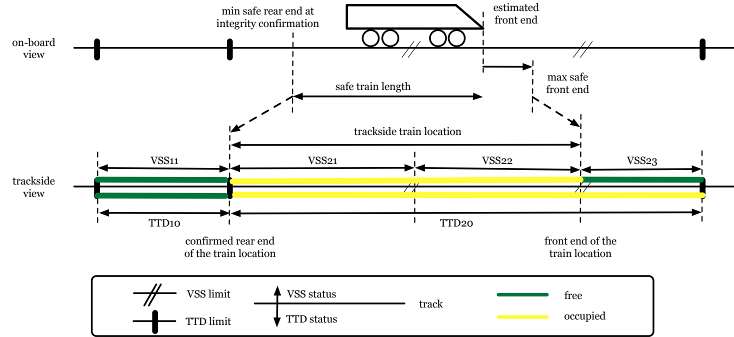

Fig. 1. Hybrid ERTMS/ETCS Level 3 System Conventions [7]

2.3 Hybrid ERTMS/ETCS Level 3 Basics

The train separation function of ERTMS/ETCS Level 3 relies entirely on the condition that the system knows at all times the position, length, and integrity status of the train [7]. Each train needs to be fitted with a Train Integrity Monitoring System (TIMS) to report its position and integrity status to the system. Due to the limitation of GSM-R communication, these pre-conditions for Level 3 operation are not satisfied as the train may disconnect from the system because of poor communication. The HL3 concept is brought up to solve the disconnect issue by using a limited implementation of track-side train detection. Trains that are disconnected from the HL3 are still visible using track-side train detection. Thus trains which are not confirming integrity can still be authorized to run on the line.

Figure 1 shows the HL3 system conventions. The track line is divided into Trackside Train Detection (TTD) sections according to the track-side equipment. If no train is shown on the TTD section, the TTD section is considered as *free*. Otherwise, it is considered as *occupied*. This large physical section is then split into as many Virtual Sub-Section (VSS) as required for the intended performance. These VSS are fixed virtual blocks to avoid train collision. The occupation status of the VSS is determined using both TTD status information and position reports of the train. The VSS is considered as *free* when the track-side is certain that no train is located on the VSS while it is considered as *occupied* when some integer train is located on this VSS while the track-side is certain that no other vehicle is located on the same VSS. Status *unknown* and *ambiguous* are used to indicate the states under the scenario with disconnected trains. A VSS is considered as *unknown* when there is no certainty if it is free. And a VSS is considered as *ambiguous* when it is known to be *occupied* but it is unsure whether there is another train on the same VSS. The track-side detection equipment can improve the system performance by providing a faster release of VSS when the TTD is *free* on the basis of train position reports. A train on a track with an established safe radio connection to the track-side is considered as a connected train. The *train location* defines the track-side view of the VSS that

is currently occupied by a connecting train, whose granularity is one VSS. The front and rear end of the train location is considered independently from each other. Each train has an estimated front end, while the rear end is derived from the estimated front end and the safe train length through train integrity confirmation. It takes time for a train to stop after it applies brakes. The estimated front end and rear end are extended to the max safe front end and min safe rear end with an additional safety margin to guarantee the safety properties of the system. When the track-side receives the report that the max safe front end of the train has entered a VSS, it considers the train to be located on this VSS. A train that allows the track-side to release VSS in the rear of the train based on its position reports is defined as integer train [7]. However, when modelling the HL3 system in Event-B is complicated as the events in Event-B models can be difficult to validate due to the complexity of conditions that are challenging to explain to domain experts. Fischer and Dghaym propose to create test cases on Event-B models using a Cucumber framework, which defines lightweight structures for describing the expected behaviour readable by both domain experts and modelers [6]. Based on their definition for the concrete scenarios, we define approaches to map concrete scenarios to abstract scenarios and refine the abstract scenarios to concrete scenarios.

3 Example Scenario

In this section, we use Scenario 4: Start of Mission / End of Mission in [7] to illustrate our approach to generation of abstract scenarios. In this scenario, there are eight numbered steps. However, since most steps contain a sequence of actions and consequent state changes, we break the steps down further into sub-steps [1]. We also note that the associated diagram (Figure 2) shows, for each step, more details about the expected state, than is given in the text. We have included some (but for brevity, not all) of this state in the scenario. Hence, the sub-steps given in *italic* are derived from the diagram rather than the original text of [7].

1. (a) Train 1 is standing on VSS 11
   (b) with desk closed and no communication session.
   (c) All VSS in TTD 10 are “unknown”.
   (d) *TTD 10 is occupied and TTD20 is free.*
2. (a) Train 1 performs the Start of Mission procedure.
   (b) Integrity is confirmed.
   (c) Because train 1 reports its position on VSS 11,
   (d) this VSS becomes ”ambiguous”.
3. (a) Train 1 receives an OS MA until end of VSS 12
   (b) and moves to VSS 12

[1] Note that we have adapted step 3 slightly compared to the specification because our model does not support granting Full Supervision Movement Authority (FS MA) containing VSS that are not free

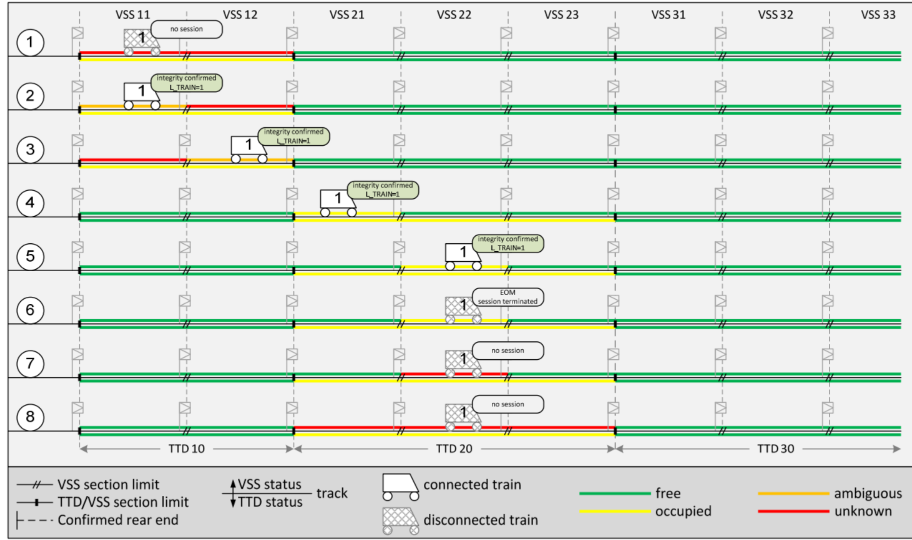

Fig. 2. Start of Mission / End of Mission [7]

   (c) which becomes "ambiguous".
   (d) VSS 11 goes to "unknown".
   (e) Train 1 receives an FS MA until end of VSS 22
4. (a) Train 1 moves to VSS 21
   (b) which becomes occupied
   (c) and all VSS in TTD 10 become "free", VSS 11 and VSS 12.
   (d) *TTD 10 is free and TTD20 is occupied.*
5. (a) Train 1 continues to VSS 22
   (b) which becomes "occupied".
   (c) VSS 21 becomes "free";
6. Train 1 performs the End of Mission (EOM) procedure.
7. (a) Due to the EoM procedure VSS 22 goes to "unknown"
   (b) and the disconnect propagation timer of VSS 22 is started.
8. (a) The disconnect propagation timer of VSS 22 expires.
   (b) All remaining VSS in TTD 20 go to "unknown"

This example scenario is useful for understanding the specification but it still contains ambiguities that are revealed when considering a formally precise model. For example trains do not usually move to a new section in one atomic step; it is not stated when position reports are sent or what information they contain. In addition, the use of natural language is not always consistent; in order to animate the scenario in a repeatable way with tool support, we need a more consistent syntax. We also need more abstract versions of the scenario if we wish to validate the initial stages of our model.

4 Domain Specific Language

To improve clarity and precision, we suggest a DSL for HL3 scenarios that aims to retain understandability for domain experts of the natural language version. We select nouns that are used in the natural language version of the scenario to describe domain objects and their state. These will be used to describe the expected state of the model. We select a set of adjectives to provide a consistent way to link the nouns when describing state. Finally we select a set of verbs to describe transitions that change the state of objects The DSL is generic in the sense that it is agnostic of the target modelling language, although very specific to the HL3 problem domain. In order to adapt the DSL for use with a particular modelling notation (in our case Event-B) cucumber step definitions must be written. Examples of these are shown in Section 7.

The kind of formal refinement modelling that we wish to support is based on abstract representation of state. In each refinement further distinction of the state values are added, either by replacing a state variable with an alternative one that gives finer detail, or by adding a completely new variable. As state details are added, the transition events that change state are elaborated to deal with the new values. In many cases completely new transitions are revealed. As the model refinement process is state driven, so is our DSL for scenario abstraction/refinement. Therefore in the DSL we add alternative names for state values so that the scenario can be adapted to abstract levels by re-phrasing clauses when the state is modelled more abstractly.

Nouns

```
1  <train> = <label>
2  <section> = TTDx
3  <sub-section> = <section>.VSSy
4  <ma>    = <abstract ma> | <concrete ma>
5  <abstract ma> = MA until <sub-section>
6  <concrete ma> = FSMA until <sub-section> | OSMA until <sub-section>
7  <timer> = <sub-section>.DisconnectTimer | <sub-section>.
       ShadowTimer | <sub-section>.GhostTimer
8  <section state> = FREE | OCCUPIED
9  <sub-section sate> = <abstract sub-section state> | <concrete sub-
       section state>
10 <abstract sub-section state> = AVAILABLE | UNAVAILABLE
11 <concrete sub-section state> = FREE| OCCUPIED | AMBIGUOUS | UNKNOWN
```

Adjectives

```
1  <train> stood at <sub-section>
2  <train> connected | disconnected
3  <train> in mission | no mission
4  <train> is integral | is split
```

```
5 <train> has <ma>
6 <section> is <section state>
7 <sub-section> is <sub-section state>
```

Verbs

```
1 <train> enters | leaves <sub-section>
2 <train> connects | disconnects
3 <train> starts mission | ends mission
4 <train> splits | couples
5 <train> receives <ma>
6 <timer> starts
7 <timer> expires
8 <train> reports position
9 <train> reports position as integral
10 <train> reports position as split
```

5 Concrete Scenario using DSL

With reference to the scenario steps listed in Section 3, we first illustrate how the natural language scenario of the specification can be expressed in our domain specific language. In the Section 6 we will show how to extract abstract scenarios that fit with our refinement levels.

Steps 1a,1b,1c and 1d give the initial starting state which becomes a `Given` clause in our language (Lines 1–6). Note that the track state is included as `Given` rather than checked by a `Then` clause because it does not necessarily follow from the train state. Step 2a is an action that, in our model, requires two distinct events which we conjoin in a `When` clause (Line 7) where Train1 *starts mission* and *connects*. Steps 2b and 2c, are performed in a single atomic reporting event in our model, giving another `When` clause (Line 8). Step 2d gives an expected consequence concerning the state of a VSS, which we check with a `Then` clause (Line 9). Step 3a grants an On Sight Movement Authority (OS MA) up to VSS 12, to the train (Line 10). Step 3b is somewhat ambiguous since trains can span more than one sub-section and therefore enter and leave them in distinct events which are not normally simultaneous. We interpret Step 3b as two consecutive steps; enter the new VSS 12 (Line 11) and then leave the previous VSS 11 (Line 12). Also, we assume that the train then reports its new position as VSS 12 (Line 13), since otherwise the Virtual Block Detector (VBD) would not know to update the VSS states as indicated in Steps 3c and 3d. Step 3 is a good example of why a more precise domain specific language is needed for describing scenarios. A similar process of interpretation is followed in the remaining steps.

```
1 Given Train1 stood at VSS11
```

```
2     And Train1 disconnected
3     And TTD10.VSS11 is UNKNOWN
4     And TTD10.VSS12 is UNKNOWN
5     And TTD10 is OCCUPIED
6     And TTD20 is FREE
7   When Train1 starts mission and Train1 connects
8   When Train1 reports position as integral
9     Then VSS11 is AMBIGUOUS
10  When Train1 receives OSMA until VSS12
11  When Train1 enters VSS12
12  When Train1 leaves VSS11
13  When Train1 reports position as integral
14    Then VSS12 is AMBIGUOUS
15    And VSS11 is UNKNOWN
16  When Train1 receives FSMA until VSS22
17  When Train1 enters VSS21
18  When Train1 leaves VSS12
19  When Train1 reports position as integral
20    Then TTD10.VSS11 is FREE
21    And TTD10.VSS12 is FREE
22    And TTD10.VSS21 is OCCUPIED
23    And TTD10 is OCCUPIED
24    And TTD20 is FREE
25  When Train1 enters VSS22
26  When Train1 leaves VSS21
27  When Train1 reports position as integral
28    Then TTD20.VSS21 is FREE
29    And TTD20.VSS22 is OCCUPIED
30  When Train1 disconnects and Train1 ends mission
31    Then TTD20.VSS22 is UNKNOWN
32    Then VSS22.disconnect_propagation_timer starts
33  When VSS22.disconnect_propagation_timer expires
34    Then VSS21 is UNKNOWN
35    And VSS23 is UNKNOWN
```

6 Abstract Scenarios

In order to obtain scenarios that can be used to validate our abstract models, we deduce correspondingly abstract scenarios from the concrete one that has been translated into our DSL in Section 5. To do this, we consider the data refinement of the model including superposition of new data. The process systematically reduces the concrete scenario by omitting any irrelevant details and only retaining clauses that relate to the data representations used in that refinement level. Note that data representation may vary in refinement levels which

affects the Cucumber step definition used to convert the scenarios into a form that can be used to animate the model.

Once a state has been checked at a particular refinement level it does not need to be checked at subsequent levels because the proof of refinement ensure this. Any Then clauses of the previous level are omitted and only if the state data representation is refined to add more detail is it necessary to add new Then clauses. In our case the concrete scenario derived from the specification has the correct final Then clauses to match our most concrete model refinement. In general the starting specification scenario could contain excess state checks that are already dealt with in earlier refinement levels. The number of Then clauses to add, is somewhat subjective; one could for example check that nothing else has changed state after each When clause. In the examples we have avoided this and adopt the same policy as the given scenario of the specification which is to only check for expected changes in state. In the rest of this section, we present how the specification scenario is abstracted at the different level of refinement according to our development.

Movement on VSS. Our most abstract model contains no other state except for the position of trains on VSS and hence, for its scenario, we pick only the clauses that are related to train movement and add Then clauses that check the train's position after each movement.

```
1 Given Train1 stood at VSS11
2 When Train1 enters VSS12
3   Then Train1 stood at VSS11,VSS12
4 When Train1 leaves VSS11
5   Then Train1 stood at VSS12
6 When Train1 enters VSS21
7   Then Train1 stood at VSS12,VSS21
8 When Train1 leaves VSS12
9   Then Train1 stood at VSS21
10 When Train1 enters VSS22
11   Then Train1 stood at VSS21,VSS22
12 When Train1 leaves VSS21
13   Then Train1 stood at VSS22
```

Radio communication and TTD. In our first and second refinements we add radio communication and status of TTD. Here we have combined them into one scenario for brevity. We add Then clauses to check train connection and TTD state after any When clause that should affect this.

```
1 Given Train1 stood at TTD10.VSS11
2   And Train1 is disconnected
3   And TTD10 is OCCUPIED
4   And TTD20 is FREE
```

```
5  When Train1 connects
6    Then Train1 connected
7  When Train1 enters TTD10.VSS12
8  When Train1 leaves TTD10.VSS11
9  When Train1 enters TTD10.VSS21
10   Then TTD20 is OCCUPIED
11 When Train1 leaves TTD10.VSS12
12   Then TTD10 is FREE
13 When Train1 enters TTD10.VSS22
14 When Train1 leaves TTD10.VSS21
15 When Train1 disconnects
16   Then Train1 disconnected
```

Introduce missions and generic movement authority. Our next model refinement introduces movement authority but does not distinguish between Full Supervision Movement Authority (FS MA) and OS MA modes. In the scenario we must use the generic form of the DSL syntax which was introduced for this purpose. Note that we still split the granting of Movement Authority (MA) into two `When` clauses so that the state check is an abstract version of the order that will later be enforced in a refinement. The refinement also introduces the start of mission and end of mission procedures.

```
1  Given Train1 stood at TTD10.VSS11
2    And Train1 disconnected
3    And TTD10 is OCCUPIED
4    And TTD20 is FREE
5  When Train1 starts mission and Train1 connects
6    Then Train1 in mission
7  When Train1 receives MA until TTD10.VSS12
8    Then Train1 has MA until TTD10.VSS12
9  When Train1 enters TTD10.VSS12
10 When Train1 leaves TTD10.VSS11
11 When Train1 receives MA until TTD20.VSS22
12   Then Train1 has MA until TTD20.VSS12
13 When Train1 enters TTD20.VSS21
14 When Train1 leaves TTD10.VSS12
15 When Train1 enters TTD20.VSS22
16 When Train1 leaves TTD20.VSS21
17 When Train1 disconnects and Train1 ends mission
18   Then Train1 no mission
```

Introduce position reports, VSS availability, integrity and distinguish between FS and OS MA. In this refinement, we we refine MA to distinguish between FS

MA and OS MA and introduce position and integrity reporting of trains which, in conjunction with TTD status, determines abstract VSS status. Notice that we replace the more abstract MA checks with OS MA and FS MA ones. At this stage, VSS status is bi-state instead of the final four states of the concrete scenario.

```
1  Given Train1 stood at TTD10.VSS11
2    And Train1 disconnected
3    And TTD10.VSS11 is UNAVAILABLE
4    And TTD10.VSS12 is UNAVAILABLE
5    And TTD10 is OCCUPIED
6    And TTD20 is FREE
7  When Train1 starts mission and Train1 connects
8  When Train1 reports position as integral
9  When Train1 receives OSMA until TTD10.VSS12
10   Then Train1 has OSMA until TTD10.VSS12
11 When Train1 enters TTD10.VSS12
12 When Train1 leaves TTD10.VSS11
13 When Train1 receives FSMA until TTD20.VSS22
14   Then Train1 has FSMA until TTD20.VSS22
15 When Train1 enters TTD20.VSS21
16 When Train1 leaves TTD10.VSS12
17 When Train1 reports position as integral
18   Then TTD10.VSS11 is AVAILABLE
19   And TTD10.VSS12 is AVAILABLE
20   And TTD10.VSS21 is UNAVAILABLE
21 When Train1 enters TTD20.VSS22
22 When Train1 leaves TTD20.VSS21
23 When Train1 reports position as integral
24   Then TTD10.VSS21 is AVAILABLE
25   And TTD10.VSS22 is UNAVAILABLE
26 When Train1 disconnects and Train1 ends mission
```

Introduce timers. This refinement introduces propagation timers that expand the unavailable area of VSS in case a non-communicative train moves. When the propagation timer expires, the adjacent VSS in the TTD become unavailable. Notice that the scenario is not like a refinement; we can add checks of old variables when further steps of the scenario should affect this. In the previous scenario we did not specify the state of these VSS,hence leaving room to add them now without introducing a contradiction.

```
1  Given Train1 stood at TTD10.VSS11
2    And Train1 disconnected
3    And TTD10.VSS11 is UNAVAILABLE
4    And TTD10.VSS12 is UNAVAILABLE
```

```
5    And TTD10 is OCCUPIED
6    And TTD20 is FREE
7  When Train1 starts mission and Train1 connects
8  When Train1 reports position as integral
9  When Train1 receives OSMA until TTD10.VSS12
10 When Train1 enters TTD10.VSS12
11 When Train1 leaves TTD10.VSS11
12 When Train1 receives FSMA until TTD20.VSS22
13 When Train1 enters TTD20.VSS21
14 When Train1 leaves TTD10.VSS12
15 When Train1 reports position as integral
16 When Train1 enters TTD20.VSS22
17 When Train1 leaves TTD20.VSS21
18 When Train1 reports position as integral
19 When Train1 disconnects
20 When Train1 ends mission
21   Then TTD20.VSS22.disconnect_propagation_timer starts
22 When TTD20.VSS22.disconnect_propagation_timer expires
23   Then TTD20.VSS21 is UNAVAILABLE
24   And TTD20.VSS23 is UNAVAILABLE
```

Introduce VSS state. In this refinement of the scenario we introduce the full VSS states of the specification. That is, available is replaced by free and not available is replaced by ambiguous, occupied or unknown as appropriate This refinement brings us back to the full concrete scenario that was described in Section 4.

7 Tool Support

In this section, we show examples of specifying step definitions that link the domain specific scenarios with our model at different levels of refinement. Our step definitions are built on top of the Cucumber for Event-B.

We start with our most abstract model which has events for trains to enter or leave a VSS. The signature of the event to move the rear of a train is as follows

```
1 event ENV_rear_leave_section
2 any
3   tr // The train
4   vss // The VSS from that the train moves
5 where ... then ... end
```

In order to link the above event with the Gherkin commands, e.g., `When Train1 leaves VSS11`, we define the following step definition.

```
1 When(~/^${id} enters ${id}$/) {
```

```
2   String train, String vss ->
3   fireEvent("ENV_rear_leave_section", "tr = " + train + " & " + "vss = " + vss
      )
4 }
```

Here fireEvent is a library method from Cucumber for Event-B to fire an event in the model with possible additional constraints on the event's parameters. In the step definition above, the information about the train ID and the VSS is extracted using pattern matching and subsequently used to build the parameter constraint accordingly.

In the same model, we have a variable occupiedBy $\in$ VSS $\leftrightarrow\!\!\!\twoheadrightarrow$ train to keep track of information about occupation of VSS by trains. We can use this to specify the step definition for commands, such as, `Then Train1 stood at VSS11,VSS12`, as follows

```
1 Then(~/^${id} stood at ${id_list}$/) {
2   String train, String vss_set ->
3   assert true == isFormula("occupiedBy ~[{" + train + "}]", "{" + vss_set +
      "}")
4 }
```

Here isFormula is a library method from Cucumber for Event-B to compare the evaluation of a formula (e.g., $\mathsf{occupiedBy}^{-1}[\{\mathsf{TRAIN1}\}]$) and the expected result (e.g., $\{\mathsf{VSS11}, \mathsf{VSS12}\}$).

Step definitions might need to change according to refinements of the model. For example, when we introduce TTD information, event ENV_rear_leave_section is split into two events: ENV_last_train_leave_ttd (when the TTD will be freed) and ENV_rear_leave_section otherwise. We introduce an alternative step definition, which selects whichever case is enabled, to reflect this refinement:

```
1 When(~/^${id} leaves ${id}$/) {
2   String train, String vss ->
3   String formula = "tr = " + train + " & " + "vss = " + vss
4   if (isEventEnabled("ENV_rear_leave_section", formula))
5     fireEvent("ENV_rear_leave_section", formula)
6   else if (isEventEnabled("ENV_last_train_leave_ttd", formula))
7     fireEvent("ENV_last_train_leave_ttd", formula)
8 }
```

8 Future Work

We have previously used natural language descriptions of scenarios manually converted ad-hoc into cucumber and executed with model animation tools. The

use of a DSL and abstract scenarios is a new proposal that requires further investigation and development. In future work we will continue to develop scenarios from the HL3 case study and investigate tool automation of the abstractions based on the refinements from the model. We will employ the scenario-based modelling techniques in other domains such as aerospace to test its generality. Our eventual aim is to utilise the scenarios in a 'kind of' continuous integration development environment for formal modelling. Our future project commitments include model transformation from Event-B systems models to semi-formal component models and the use of precise and abstract scenarios could be utilised to validate and verify this transformation stage by co-simulation of scenarios in both models.

9 Conclusion

One of the strengths of formal methods lies in efficient, generic verification (using theorem provers) which obviates the need for test cases and hence instantiation with objects. However, to leverage this strength we need to convince domain experts and, of course, ourselves, of the validity of the models. To this end we have returned to a strategy analogous to testing; animation of models using scenarios. We envisage a growing reliance on scenarios as we seek to integrate formal systems level modelling with industrial development processes. An important step is to make the scenarios more precise so that they are clear and unambiguous while remaining easily understood by all stakeholders. We have suggested using an easily derived DSL to achieve this. For early detection of problems, it is important that we can use the scenarios at stages when our abstract models do not contain all of the detail involved in the concrete scenario. We therefore propose a technique of synthesising abstract versions of the scenario that are suitable for use with the abstract refinement levels of the model. The abstraction technique uses the data refinement of the model (including superposition of new data as well as refinement of data representation) to make corresponding abstractions in scenarios. We propose to develop these techniques in the future as we continue to build our formal model based development process.

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. *STTT*, 12(6):447–466, 2010.
3. Jacob L Cybulski. The formal and the informal in requirements engineering. Technical report, Technical Report 96/7, Department of Information Systems, The University of . . . , 1996.
4. Dana Dghaym, Michael Poppleton, and Colin Snook. Diagram-led formal modelling using iuml-b for hybrid ertms level 3. In *6th International ABZ Conference ASM, Alloy, B, TLA, VDM, Z, 2018, Proceedings of*, 2018.

5. Tomas Fischer. Cucumber for Event-B and iUML-B. https://github.com/tofische/cucumber-event-b, 2018.
6. Tomas Fischer and Dana Dghaym. Formal model validation through acceptance tests. In *RSSRail 2019*, 2019.
7. EEIG ERTMS Users Group. Hybrid ERTMS/ETCS Level 3:Principles, July 2017. Ref. 16E042 Version 1A.
8. Thai Son Hoang. An introduction to the Event-B modelling method. In *Industrial Deployment of System Engineering Methods*, pages 211–236. Springer-Verlag, 2013.
9. 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.
10. Matt Wynne and Aslak Hellesøy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Programmers, LLC, 2012.