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, Asieh Salehi Fathabadihttps://orcid.org/0000-0002-0508-3066, and Michael Butlerhttps://orcid.org/0000-0003-4642-5373

ECS, University of Southampton, Southampton, U.K.,
{`cfs, t.s.hoang, d.dghaym, a.salehi-fathabadi, mbutler`}`@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 whether the scenario is feasible in the model and that the model reaches the states expected in the scenario. However, there are two problems with this approach. 1) 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. 2) Provided scenarios are typically at the most concrete level corresponding to the full requirements and cannot be used until all the refinements have been completed in the model. We 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) and can be used as the persistence for automated tool support. We propose two alternative approaches to using scenarios during formal modelling: A method of refining scenarios before the model is refined so that the scenarios guide the modelling, and a method of abstracting scenarios from provided concrete ones so that they can be used to test early refinements of the model. We illustrate the two approaches on the 'Tokeneer' secure enclave example and the ERTMS/ETCS Hybrid Level 3 specification for railway controls. We base our approach on the Cucumber framework for scenarios and the Event-B modelling language and tool set. We have developed a new 'Scenario Checker' plugin to manage the animation of scenarios. [1].

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

[1] The example model and scenario scripts supporting this paper are openly available at `https://doi.org/10.5258/SOTON/D1026`

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 [9] 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 [21,6] 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. Our experience in using scenarios [10] led us towards the use of Domain Specific Language (DSL) to specify scenarios more precisely [20]. Here we bring together ideas from [20] and [6] as well as new contributions for scenario-refinement based methods, and propose a general approach to using domain-specific scenarios for formal refinement-based methods.

In some cases the modellers may need to construct scenarios as part of the modelling process and refine them in parallel with development of the model. The scenarios provide an interactive exploration of behaviour so that the modeller can assess whether the model satisfies the requirements via iterative development cycles. We describe our experience of using a scenario-refinement based approach to drive the formal modelling using the Tokeneer case study [3]. In other cases, scenarios may be given to the modeller as part of the system requirements. For this case we propose a technique of synthesising abstract scenarios from given 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 this approach, called scenario-abstraction approach, using the European Rail Traffic Management System (ERTMS)/European Train Control System (ETCS), Hybrid Level 3 (HL3) specification [13] for which we have previously developed a formal model presented in [10]. In both approaches we advocate the use of a 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.

The paper is structured as follows: Section 2 discusses related and previous work. Section 3 introduces the HL3 and Tokeneer case studies used to illustrate the paper. Section 4 provides background on the Event-B formal modelling language, UML-B and the Cucumber framework for scenarios. Section 5 introduces the idea of using scenarios in formal modelling, discusses different types of scenarios and gives the motivation for inventing a DSL for each problem domain. It concludes by summarising two alternative approaches; scenario-refinement and scenario-abstraction. Section 6 reports our experience of using the first, scenario-refinement approach, including how to design and refine scenarios in parallel with modelling the system through refinements. This section is illustrated with examples from the Tokeneer case study. Section 7 explains an alternative scenario-abstraction approach where the concrete scenarios are provided as input to the modelling stage and need to be abstracted to be useful. It discusses how the DSL and scenarios support refinement. This section is illustrated with examples from the HL3 case study. Section 8 describes future work and Section 9 concludes.

2 Related Work

Scenarios in general: User requirements and system behaviour are often explained using scenarios. Requirement-level scenarios can be adopted as a primary design artifact and refined through detailed scenarios. Carroll et al. [8] classify scenarios according to their use in systems development ranging from requirements analysis, user-designer communication, software design, through to implementation, training and documentation. In formal modelling, scenarios can be used alongside formal verification to test models. Iliasov [14] developed a language (Flow) to describe use cases, with features to make assertions about event enabled-ness and feasibility. The main difference to our work is that Iliasov focused on *verification using proof*, e.g., generating and proving verification conditions to ensure that the models satisfy the use cases expressed using the Flow language. We focus on *validation using simulation* by expressing scenarios using DSLs and automatically executing the scenarios on the models. Our automatic execution framework extends that proposed by Fischer and Dghaym [12] who used the Cucumber framework [23] to create executable scenarios for Event-B models.

DSLs and Scenarios: James and Roggenbach [15] propose the use of DSLs as a way to aid in the uptake of formal methods within industry. They illustrate the methodology within the railway domain using the algebraic specification language CASL. The methodology demonstrates formalising a DSL through automatic translation of a UML class diagram followed by proving domain specific lemmas and developing a graphical editor for the DSL. Bodeveix et al. [5] use DSLs with the B formal method for the development of process schedulers in Linux and the Chorus real-time operating system. Carioni et al. [7] present an ASM based DSL for scenario-based validation (the AValLa language) and its supporting tool (the AsmetaVvalidator). The *actor* concept of UML use-cases is extended with the concept of an *observer actor*; an external observer who can interact with the system model and check the system state. Using AValLa, the abstract syntax of the DSL language is defined by a meta-model and then a concrete notation is automatically derived from the abstract syntax. The DSL semantics are given in terms of ASMs. AValLa is supported by the AsmetaV (ASM Validator) tool to execute AValLa scenarios.

Scenario Refinement: Sobernig et al. [22] outline the role of executable abstract scenarios which are capable of being refined to concrete scenarios, through a scenario-based meta-model testing approach. Arcaini and Riccobene [4] propose automatic refinement of scenarios for testing Abstract State Machines (ASMs). They emphasise reusability of formal development artefacts that can be achieved by adapting scenarios from abstract levels to refinements. The approach they apply for generating refined scenarios is based on the classical model checking technique of test generation. In this case the authors build an LTL formula from the abstract scenario, negate it and then use the AsmetaSMV model checker to find a counter example in the refined model. The counter example is then used to extract a scenario for the refined model. Although this will indeed generate a refined scenario from an abstract one, it is not enough to fully exercise the new behaviour introduced during refinement. The authors confirm this and suggest that the modeller might need to develop new scenarios from scratch to test new behaviours. This is equivalent to the guided development of refined scenarios in our scenario-refinement approach. Malik et al. [16] manually devise abstract scenario which are then refined in parallel with Event-B models. The scenarios are represented using Communicating Sequential Process (CSP) expressions, which can be refined in conformance with the Event-B models. The authors use a restricted style of refinement which enables automatic derivation of the refined CSP scenarios. Our approach is more generic because we do not restrict the refinement process.

Summary: By comparison, we propose using problem specific DSLs for expressing scenarios with links to tool support for scripted or user controlled executions. We support the scenario-refinement approach suggested by other authors but also suggest using scenario-abstraction approach when concrete scenarios are already available as part of the system description.

3 Case Studies

3.1 Hybrid ERTMS/ETCS Level 3 specification

The Hybrid ERTMS/ETCS Level 3 (HL3) specification [13] describes a system for controlling trains moving on a linear track and communicating by radio and trackside equipment. The system is designed to allow for the presence of older trains that are only detected by trackside sensors. A train movement controller manages the Movement Authority (MA) granted to each train that it is in communication with. The focus of the specification is the Virtual Block Detector (VBD), which conservatively estimates train locations to a finer granularity than physically detected track sections, and thus reports free virtual track subsections (Virtual Sub-Section (VSS)) available for train movement. Trains and trackside report location data to the VBD which then reports free track sections to the train movement controller. The MA granted to each train consists of a set of sections that the train is permitted to move into.

The train separation function of HL3 relies entirely on the condition that the system knows at all times the position, length, and integrity status of the train. 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 always 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

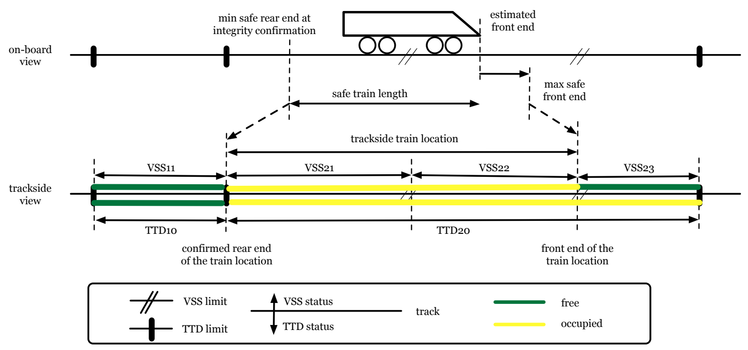


Fig. 1: Hybrid ERTMS/ETCS Level 3 System Conventions [13]

disconnected from the HL3 are still visible using track-side train detection. Thus trains that 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-Sections as required for the intended performance. These Virtual Sub-Sections 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 an integer train [13].

3.2 Tokeneer Secure Enclave

The Tokeneer system [3] consists of a secure enclave and a set of system components, some housed inside the enclave and some outside (Figure 2). In this paper, we focus on modelling the system-level requirements which prevent users from entering the enclave if they do not hold appropriate permission. The ID Station interfaces to four different physical devices: fingerprint reader, smartcard reader, door and visual display. The primary objective is to prevent unauthorised access to the Secure Enclave. The requirements include (1) authenticating individuals for entry into an enclave and (2) controlling the entry to and egress from an enclave of authenticated individuals. The door has four possible states: the cross-product of *open*/*closed* and *locked*/*unlocked*. In order to prove the security property we have to make an additional assumption that no tail-gating is possible. To do this we only allow one person at a time to approach the door and they may only do so when it is closed and locked. A card identifies a particular user using a fingerprint mechanism. If a user holds a card that identifies themselves, they are permitted in the enclave. Hence cards should only be issued to permitted users. A successful scenario involves: arrival of a permitted user at the door who then presents a card on the card reader and a matching finger print at the fingerprint reader. The system will then unlock the door allowing the user to open it and enter the enclave.

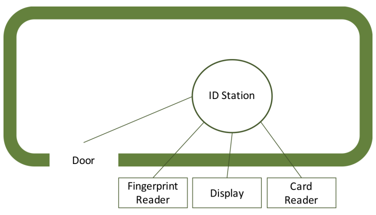


Fig. 2: Tokeneer Secure Enclave

4 Background

4.1 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* $\mathsf{A}(\mathsf{c})$ 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.

For example the enterEnclave event from the Tokeneer case study described in Section 3.2, has one parameter this_user and two guards. The first guard grd1 checks whether this_user is already in the enclave, while grd2 checks if this_user is permitted in the enclave. If both guards are satisfied then enterEnclave can be executed and the action act will update the variable inEnclave by adding a new instance this_user.

```
event enterEnclave
any this_user
where
@grd1: this_user ∉ inEnclave
@grd2: permittedInEnclave(this_user)=TRUE
then
@act: inEnclave := inEnclave ∪ {this_user}
end
```

Each machine also has a special event called INITIALISATION, with no parameters or guards, that puts 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 or machine refinement.

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

4.2 UML-B

UML-B [17,18,19] 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 UML-B tools. State-machines are typically refined by adding nested state-machines to states. Class diagrams provide a way to visually model data relationships.

4.3 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 that can be executed on the specifications of a system. BDD is important for communication between the business stakeholders and the software developers. Gherkin/Cucumber [23] is one of the various frameworks supporting BDD. Gherkin [23, 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 [21], we described our specialisation of Cucumber for Event-B with the purpose of automatically executing scenarios for Event-B models. Cucumber is a framework for executing acceptance tests written in the Gherkin language and provides a Gherkin language parser, test automation as well as report generation. We provide Cucumber step definitions for Event-B in [11] 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.

5 Scenarios

In this section we discuss different types of scenarios, explain the motivation for providing a DSL and introduce two alternative approaches to using scenarios in formal modelling. This section is based mostly on the HL3 specification which contains explanatory scenarios that illustrate the motivation for using a DSL.

5.1 Types of Scenarios

We identify two orthogonal categorisations of scenarios. A conceptual categorisation of scenarios is that they may be seen as nominal or non-nominal depending on their relevance to the main purpose of the system. A more practical categorisation is whether they are positive (something should happen) or negative (something should not happen).

– **Nominal scenarios** are designed to exercise the main successful behaviours of the system.
– **Non-nominal scenarios** are alternative outcomes where a nominal scenario does not succeed. They can be deduced by finding deviations from the nominal ones that are caused by conditions in the nominal scenario not being met. Non-nominal scenarios work in exactly the same way to nominal ones since they represent valid possible sequences of events and responses. They only differ in our conception of what normal behaviour is and sometimes this can be subjective. Typically, non-nominal scenarios are added in refinements when details for the nominal scenarios to succeed are elaborated.
– **Positive scenarios**: can be expressed as a sequence of transition events that should be possible in the system. The scenario tests that a sequence of events, with particular parameter values, is feasible (i.e. that events are enabled).
– **Negative scenarios**: we also consider scenarios which we do not want to be possible in the system. These negative scenarios involve a check that some particular events are disabled at a particular state of the system. Note that disabledness is preserved by refinement since guards must not be weakened in refinement. Hence, as for data checks performed in **When** clauses, we do not need to re-check disabledness of events in negative scenarios. On the other hand, standard Event-B refinement does not guarantee enabledness which must, therefore, always be checked using a **When** clause to make sure it is re-checked in refinements by the refined scenarios.

In the case of Tokeneer, a permitted user succeeding to enter the enclave is a positive nominal scenario. A permitted user may approach the enclave but decide not to enter the enclave after the door unlocks and opens, this as a positive non-nominal scenario A non-permitted user should not be able to enter the enclave, this is a negative nominal scenario since the scenario needs to check that the event of entering is disabled. These examples are shown in more detail in Section 6

5.2 Example Scenario

In this section, we use Scenario 4: Start of Mission / End of Mission in [13] 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 [2]. We also note that the associated diagram (Figure 3) shows, for each

[2] 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

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 *italics* are derived from the diagram rather than the original text of [13].

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
   (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.

5.3 Domain Specific Language

To improve clarity and precision, we suggest a DSL (Figure 4) 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

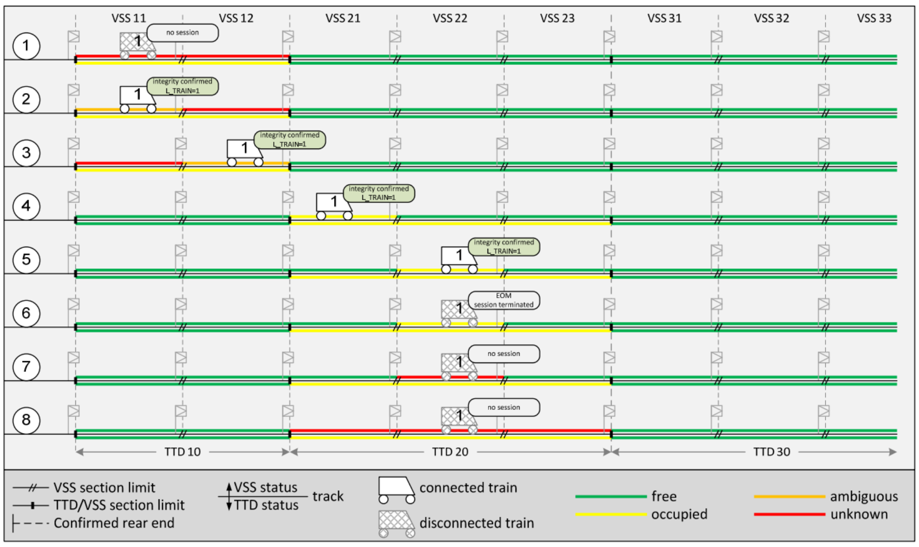


Fig. 3: Start of Mission / End of Mission [13]

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 **??**. The process of constructing the DSL and adapting it using cucumber step definitions is straightforward and relatively quick compared to the modelling stage. Hence, a new DSL can be invented for each specification domain before beginning to construct a formal model of it.

The kind of formal refinement modelling that we wish to support is based on an 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

```
<train> = <label>
<section> = TTDx
<sub-section> = <section>.VSSy
<ma>    = <abstract ma> | <concrete ma>
<abstract ma> = MA
<concrete ma> = FSMA | OSMA
<timer> = <sub-section>.DisconnectTimer | <sub-section>.ShadowTimer | <sub-section>.GhostTimer
<section state> = FREE | OCCUPIED
<sub-section sate> = <abstract sub-section state> | <concrete sub-section state>
<abstract sub-section state> = AVAILABLE | UNAVAILABLE
<concrete sub-section state> = FREE| OCCUPIED | AMBIGUOUS | UNKNOWN
```

Adjectives

```
<train> stood at <sub-section>
<train> connected | disconnected
<train> in mission | no mission
<train> is integral | is split
<train> has <ma>
<section> is <section state>
<sub-section> is <sub-section state>
<ma> until <sub-section>
```

Verbs

```
<train> enters | leaves <sub-section>
<train> connects | disconnects
<train> starts mission | ends mission
<train> splits | couples
<train> receives <ma>
<timer> starts
<timer> expires
<train> reports position
<train> reports position as integral
<train> reports position as split
```

Fig. 4: DSL for HL3 scenarios

5.4 Concrete Scenario using DSL

We first illustrate how the natural language scenario of the specification, listed in Section 5.2, can be expressed in our domain specific language (Figure 5). In Section 7 we will show how to extract abstract scenarios that fit with our refinement levels.

With reference to the scenario steps listed in Figure 5(a), steps 1a,1b,1c and 1d give the initial starting state which becomes a `Given` clause in our language (Lines 1–6 of Figure 5(b)). 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 VBD would not know to update the VSS states

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
   (c) which becomes ”ambiguous”.
   (d) VSS 11 goes to ”unknown”.
   (e) Train 1 receives an FS MA until end of VSS 22

(a) Concrete scenario in natural language

```
Given Train1 stood at TTD10.VSS11
And Train1 disconnected
And TTD10.VSS11 is UNKNOWN
And TTD10.VSS12 is UNKNOWN
And TTD10 is OCCUPIED
And TTD20 is FREE
When Train1 starts mission and Train1
    connects
When Train1 reports position as integral
Then TTD10.VSS11 is AMBIGUOUS
When Train1 receives OSMA until TTD10.VSS12
When Train1 enters TTD10.VSS12
When Train1 leaves TTD10.VSS11
When Train1 reports position as integral
Then TTD10.VSS12 is AMBIGUOUS
And TTD10.VSS11 is UNKNOWN
```

(b) Concrete scenario using DSL

Fig. 5: Translation of given scenario into DSL

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.

5.5 Running Scenarios

To run the scenarios we use the ProB model checker in single step animation mode. For the validation stage we need to observe the model’s behaviour as the scenario is executing. 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 The tool provides two areas of functionality that make scenario checking feasible: run to completion and record/replay.

In Event-B 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. 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. The scenario checker can be used in a first iteration to record and save a scenario which is then replayed and modified if the model has to be changed.

An alternative way to run the scenarios is to automatically execute them without human intervention using Cucumber for Event-B. This method also uses ProB to animate the model but the Cucumber script drives the execution and checks results. This method is useful as a regression test of the model after changes have been made guiding the tester to focus on scenarios which need to be revised. To automate a scenario written in our DSL we need to provide the correspondence between DSL steps and Cucumber for Event-B steps. 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

```
event ENV_rear_leave_section
any
 tr // The train
 vss // The VSS from that the train moves
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.

```
When(~/^${id} leaves ${id}$/) {
 String train, String vss ->
 fireEvent("ENV_rear_leave_section", "tr = " + train + " & " + "vss = " + vss)
}
```

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 $\mathsf{occupiedBy} \in \mathsf{VSS} \leftrightarrow\!\!\!\!\twoheadrightarrow \mathsf{train}$ 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

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

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

```
When(~/^${id} leaves ${id}$/) {
 String train, String vss ->
 String formula = "tr = " + train + " & " + "vss = " + vss
 if (isEventEnabled("ENV_rear_leave_section", formula))
  fireEvent("ENV_rear_leave_section", formula)
 else if (isEventEnabled("ENV_last_train_leave_ttd", formula))
  fireEvent("ENV_last_train_leave_ttd", formula)
}
```

6 Scenario-refinement Approach

In the HL3 case study described in Section 5, the scenarios were given as part of the system requirements. We will return to this example in Section 7 to show how a given concrete scenario can be used to validate abstract models. In this section, we describe a process for driving the modelling by constructing scenarios from the requirements to validate each refinement level. We are compelled to find abstract scenarios since at early modelling stages we have no use for concrete scenarios. This section is based on the Tokeneer system (see Section **??**) which only defines a few scenario requirements declaratively, leaving us to elaborate their content.

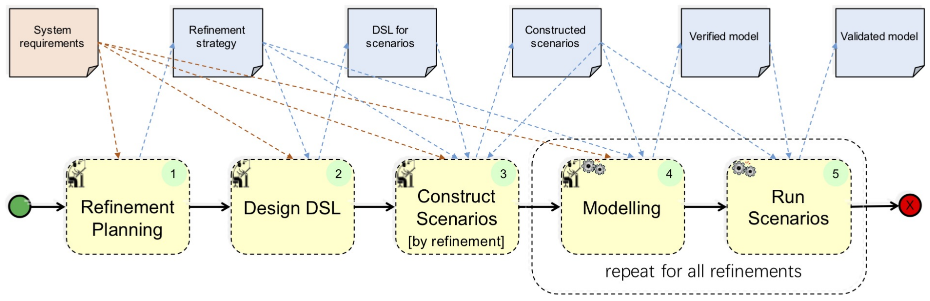


Fig. 6: Scenario-refinement process

The scenario-refinement approach is illustrated in Figure 6. Note that, although not shown, the whole process will be iterative. In our experience it is usual to revise the refinement plan during development as the models give a better understanding of issues inherent in the system.

- **Refinement plan** The first step (as is usual) is to construct a refinement plan. This gives an outline for how the model will be developed, selecting important properties to be verified at each refinement and how the details of the system will be incorporated in manageable steps. In our process the refinement plan also guides the scenarios needed at each refinement stage.

– **DSL** A DSL is constructed as described in Section 5 and taking into account any data refinements involved in refinement plan.
– *For each refinement level:*
  - **Construct scenarios** Scenarios are chosen in accordance with the guidelines of Section 5. For a refinement, new scenarios may be added which have no equivalent scenario in the abstraction. Alternatively, scenarios may be derived from abstract counterparts that exist for the previous refinement level. The model checker can be used to guide scenario refinement by showing options for traces to the next abstract step. The Scenarios are written in the DSL.
  - **Modelling** The scenarios are used to guide the construction of the next refinement of the model. The automatic theorem provers attempt to verify the model. If the automatic theorem provers do not succeed the proof obligation may reveal a mistake in the model. However, if the reason can not be determined easily, it may be more efficient to continue with the next step and check that the model is valid before investing time in manually discharging proofs.
  - **Run scenarios** The scenarios are used to exercise the model to check that it behaves as intended.

Our **refinement plan** starts with a simple model sufficient to express the security property that only permitted people are able to enter the enclave. The first refinement introduces the door locking and opening mechanisms. The door is only unlocked for permitted people. To prove the refinement we had to make a 'no tailgating' assumption that only one person is at the door and no others can approach while the door is open or unlocked. However the need for this was revealed by formal proof rather than behavioural scenarios. As a result of running some scenarios on the first refinement, we discovered the need to add a time delay to the lock mechanism to make sure there is time for a person to open the door after it is unlocked by the system. This is revealed by our scenario checker because it runs internal events such as lock and unlock automatically until no internal events are enabled. The second refinement introduces the mechanism (cards) that demonstrates that the holder is permitted in the enclave. In case a card is stolen it immutably identifies the person it permits. The third refinement elaborates the identification mechanism as being fingerprints, i.e. a card identifies the person by holding a copy of their fingerprint.

The **DSL** (Figure 7) provides a notation for users to a) enter the enclave (abstract model), b) by opening the door once it is unlocked by the system (refinement 1), c) after presenting a card that identifies themselves (refinement 2).

6.1 Constructing Scenarios by refinement

For the first abstract level of the model a set of scenarios is chosen to test the requirements that have been allocated in the refinement plan. For subsequent refinement levels, scenarios can be derived from the abstract ones by adding

Nouns

```
<user> = <label>
<location> = ENCLAVE | APPROACH | ELSEWHERE
<door> = Door
<door state> = OPEN | CLOSED
<door latch> = Latch
<latch state> = LOCKED | UNLOCKED
<card> = <label>
```

Adjectives

```
<user> is in <location>
<user> is not in <location>
<door> is <door state>
<door latch> is <latch state>
<user> holds <card>
<card> identifies <user>
<user> is allowed in <location>
```

Verbs

```
<user> enters <location>
<door> opens
<door> closes
<door latch> locks
<door latch> unlocks
<user> is_issued <card>
<user> steals <card>
<user> inserts <card>
validate <card> for <user>
```

Fig. 7: DSL for Tokeneer Scenarios

intermediate steps to exercise the added functionality. Several choices may be available for the intermediate path so that the scenario refinement relationship is one to many. The scenario is also refined by removing **Then** clauses that are guaranteed by proof and adding new **Then** clauses to check the state of new variables and enabledness of new events. Scenario construction can either be completed for all refinement levels before starting modelling or it can be interleaved with model refinements. The advantage of the former is that it provides an effective review of the refinement plan and DSL before committing to modelling.

An alternative way to find refined scenarios (similar to that suggested by Arcaini and Ricobene [4] for ASM) is to use a model checker. To do this the abstract scenario is executed on the refined model until a step is not enabled due to some new intermediate steps of the refinement. The ProB model checker is then used to search for a trace to a state where the next abstract step is enabled (by searching for a violation of the negation of its guard). We propose to investigate this tool-assisted method as an additional post-modelling validation stage in our process. It would allow us to explore generated variations of the abstract scenarios to detect any undesirable behaviour that has not been revealed by the manually constructed ones.

To illustrate the design of scenarios and their refinements, we show some examples from the Tokeneer case study using the different categorisations introduced in Section 5.1.

In the positive, nominal scenario; a permitted user succeeding in entering the enclave, the abstract version (Figure 8(a)) of the scenario simply checks that the permitted user enters the enclave and retains the permission. The first refinement (Figure 8(b)) adds checks of the door and latch states and inserts events that unlock and open the door. The second refinement (Figure 8(c)) replaces checks that the user has permission with checks that they hold a card that identifies them.

```
Given USER1 is not in ENCLAVE
And USER1 is allowed in ENCLAVE
When USER1 enters ENCLAVE
Then USER1 is in ENCLAVE
And USER1 is allowed in ENCLAVE
```

(a) m0

```
Given USER1 is not in ENCLAVE
And Door is CLOSED
And Latch is LOCKED
And USER1 is allowed in ENCLAVE
When USER1 enters APPROACH
Then USER1 is allowed in ENCLAVE
When Latch unlocks
When Door opens
When USER1 enters ENCLAVE
When Door closes
When Latch locks
Then Door is CLOSED
And Latch is LOCKED
```

(b) m1

```
Given USER1 is not in ENCLAVE
And Door is CLOSED
And Latch is LOCKED
And USER1 holds Card1
And Card1 identifies USER1
When USER1 enters APPROACH
Then USER1 holds Card1
And Card1 identifies USER1
When Latch unlocks
When Door opens
When USER1 enters ENCLAVE
When Door closes
When Latch locks
Then USER1 holds Card1
And Card1 identifies USER1
```

(c) m2

Fig. 8: Nominal scenario - Permitted user enters enclave

The positive, non-nominal scenario; a permitted user may approach the enclave but decide not to enter the enclave after the door unlocks and opens, is very similar to the previous one apart from the users action after the door opens (Figure 9(a), Figure 9(b), Figure 9(c)).

```
Given USER1 is not in
      ENCLAVE
And USER1 is allowed in
      ENCLAVE
Then USER1 is not in ENCLAVE
And USER1 is allowed in
      ENCLAVE
```

(a) m0

```
Given USER1 is not in ENCLAVE
And Door is CLOSED
And Latch is LOCKED
And USER1 is allowed in ENCLAVE
When USER1 enters APPROACH
Then USER1 is allowed in
      ENCLAVE
When Latch unlocks
When Door opens
When USER1 enters ELSEWHERE
When Door closes
When Latch locks
Then Door is CLOSED
And Latch is LOCKED
```

(b) m1

```
Given USER1 is not in ENCLAVE
And Door is CLOSED
And Latch is LOCKED
And USER1 holds Card1
When USER1 enters APPROACH
Then USER1 holds Card1
And Card1 identifies USER1
When Latch unlocks
When Door opens
When USER1 enters ELSEWHERE
When Door closes
When Latch locks
Then USER1 holds Card1
And Card1 identifies USER1
```

(c) m2

Fig. 9: Non-nominal scenario - Permitted user does not enter enclave

For the negative nominal scenario; a non-permitted user should not be able to enter the enclave (Figure 10(a)), the scenario checks that the enters ENCLAVE event is disabled for User1. (To allow for negative checking we propose to add an extra keyword Disabled to Gherkin to assert that a certain action is not allowed to take place, and Not keyword to denote the negation of an assertion). In the first refinement (Figure 10(b)) the scenario needs to check that a different event is disabled since it is now the Latch unlocks event that leads to entering the Enclave. In the second refinement we show two alternative scenarios that are both valid refinements of the scenario. In Figure 10(c)), the user holds no cards and in Figure 11 the user steals a card but it does not identify the user.

```
Given USER1 is not in ENCLAVE
And USER1 is not allowed in ENCLAVE
Then Disabled(USER1 enters ENCLAVE)
And USER1 is not in ENCLAVE
And USER1 is not allowed in ENCLAVE
```

(a) m0

```
Given USER1 is not in ENCLAVE
And Door is CLOSED
And Latch is LOCKED
And USER1
  is not allowed in ENCLAVE
When USER1 enters APPROACH
Then USER1
  is not allowed in ENCLAVE
Then Disabled(Latch unlocks)
And Door is CLOSED
And Latch is LOCKED
```

(b) m1

```
Given USER1 is not in ENCLAVE
And Door is CLOSED
And Latch is LOCKED
And USER1 holds no cards
When USER1 enters APPROACH
Then USER1 holds no cards
```

(c) m2

Fig. 10: Negative scenario - Non-permitted user cannot enter enclave

```
Given USER1 is not in ENCLAVE
And Door is CLOSED
And Latch is LOCKED
When USER1 steals Card1
And Not(Card1 identifies User1)
When USER1 enters APPROACH
And USER1 holds Card1
Then Not(Card1 identifies User1)
```

Fig. 11: Negative scenario - Non-permitted user steals card and cannot enter enclave - m2

6.2 Running the Tokeneer Scenarios

At each refinement level, our Tokeneer models were verified by proof before running any of the scenarios. Figure 12 shows the UML-B class diagram for the second refinement which introduces cards for permitted users. The provers provide insight into the system and its limitations (e.g. the danger of tailgating) leading the modeller to either add requirements or explicitly state assumptions that must be dealt with elsewhere. However, it is possible to overly restrict the behaviour of the model in order to prove properties or to leave out desired behavioural restrictions that are not needed for the properties to hold. Running the scenarios allows us to check that required behaviour is exhibited by the

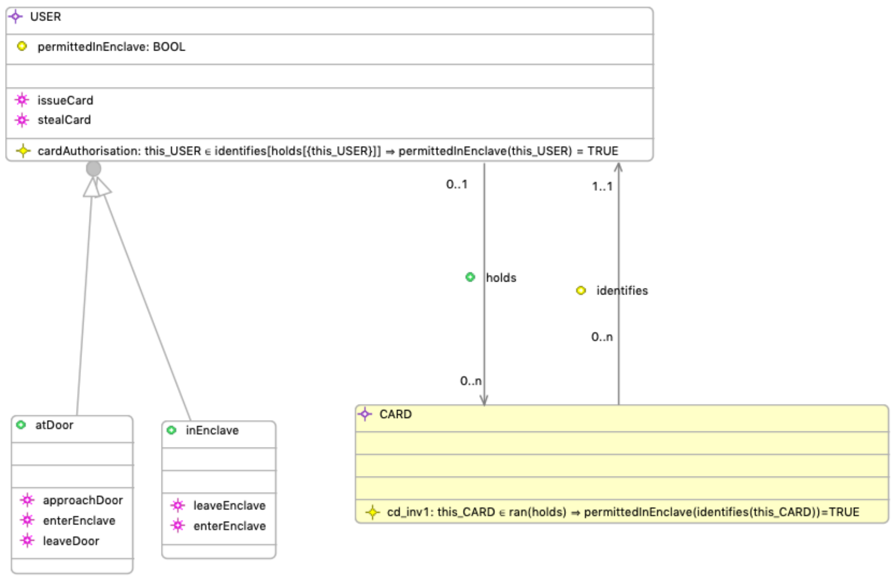

USER
permittedInEnclave: BOOL
issueCard
stealCard
cardAuthorisation: this_USER ∈ identifies[holds[{this_USER}]] ⇒ permittedInEnclave(this_USER) = TRUE
0..1
1..1
holds
identifies
0..n
0..n
atDoor
approachDoor
enterEnclave
leaveDoor
inEnclave
leaveEnclave
enterEnclave
CARD
cd_inv1: this_CARD ∈ ran(holds) ⇒ permittedInEnclave(identifies(this_CARD))=TRUE

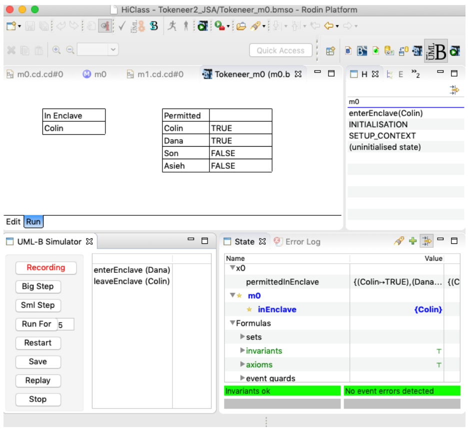
Fig. 12: UML-B class diagram for second refinement - m2

model and in doing so also provides further insight leading to iterative changes to the models and the scenarios. When running the first nominal scenario on the abstract model (Figure 13), although the scenario was executed successfully, the tester noticed that the same user can enter the enclave even when they are already inside of it. This is not physically possible without first leaving the enclave. We changed the model to incorporate this assumption and also developed a negative scenario to check that the unwanted behaviour is not possible (i.e. that enter is disabled for a user that is inside the enclave). It is important that this early stage of scenario checking is done manually by the modeller so that they can subjectively validate sensible behaviour as well as verify that the model satisfies the scenario.

When running the nominal scenario on the first refinement we wanted to make the latch unlock and lock events automatic since they are internal events of the control system. (This is an optional feature of our scenario checker tool). However, since a lock event was immediately enabled after an unlock event, the internal events do not converge to completion. This observation led us to understand that there needs to be a time delay after unlocking which we added using nested states in the UNLOCKED state (Figure 15).

7 Scenario-abstraction Approach

In this section we illustrate the scenario-abstraction approach where a concrete scenario is provided as part of a specification and abstract scenarios need to


HiClass - Tokeneer2_JSA/Tokeneer_m0.bmso - Rodin Platform

In Enclave
Colin

Permitted	
Colin	TRUE
Dana	TRUE
Son	FALSE
Asieh	FALSE

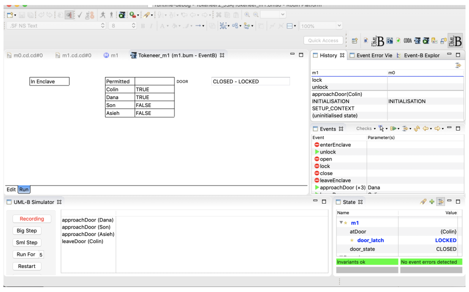
m0
enterEnclave(Colin)
INITIALISATION
SETUP_CONTEXT
(uninitialised state)

enterEnclave (Dana)
leaveEnclave (Colin)

Name	Value
x0	
permittedInEnclave	{(Colin↦TRUE),(Dana...
m0	
inEnclave	{Colin}
Formulas	
sets	
invariants	T
axioms	T
event guards	

Invariants ok
No event errors detected

Fig. 13: Running nominal scenario on abstract model m0


Tokeneer_m1 (m1.bum - EventB)

In Enclave

Permitted	
Colin	TRUE
Dana	TRUE
Son	FALSE
Asieh	FALSE

DOOR CLOSED - LOCKED

m1	m0
lock	
unlock	
approachDoor(Colin)	
INITIALISATION	INITIALISATION
SETUP_CONTEXT	
(uninitialised state)	

Event	Parameter(s)
enterEnclave	
unlock	
open	
lock	
close	
leaveEnclave	
approachDoor (×3)	Dana

approachDoor (Dana)
approachDoor (Son)
approachDoor (Asieh)
leaveDoor (Colin)

Name	Value
m1	
atDoor	{Colin}
door_latch	LOCKED
door_state	CLOSED

Invariants ok
No event errors detected

Fig. 14: Running nominal scenario on first refinement m1

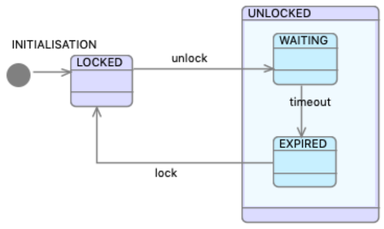


Fig. 15: UML-B state-machine for the lock behaviour in m1

be extracted to validate abstract versions of the formal model. This section is based on the HL3 specification. 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 (Fig. 16) as described in Section 5.4. 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.

```
Given Train1 stood at TTD10.VSS11
And Train1 disconnected
And TTD10.VSS11 is UNKNOWN
And TTD10.VSS12 is UNKNOWN
And TTD10 is OCCUPIED
And TTD20 is FREE
When Train1 starts mission and Train1 connects
When Train1 reports position as integral
Then TTD10.VSS11 is AMBIGUOUS
When Train1 receives OSMA until TTD10.VSS12
When Train1 enters TTD10.VSS12
When Train1 leaves TTD10.VSS11
When Train1 reports position as integral
Then TTD10.VSS12 is AMBIGUOUS
And TTD10.VSS11 is UNKNOWN
When Train1 receives FSMA until TTD20.VSS22
When Train1 enters TTD20.VSS21
When Train1 leaves TTD10.VSS12
When Train1 reports position as integral
Then TTD10.VSS11 is FREE
And TTD10.VSS12 is FREE
And TTD20.VSS21 is OCCUPIED
And TTD10 is OCCUPIED
And TTD20 is FREE
When Train1 enters TTD20.VSS22
When Train1 leaves TTD20.VSS21
When Train1 reports position as integral
Then TTD20.VSS21 is FREE
And TTD20.VSS22 is OCCUPIED
When Train1 disconnects and Train1 ends mission
Then TTD20.VSS22 is UNKNOWN
Then TTD20.VSS22.disconnect_propagation_timer starts
When TTD20.VSS22.disconnect_propagation_timer expires
Then TTD20.VSS21 is UNKNOWN
And TTD20.VSS23 is UNKNOWN
```

Fig. 16: Concrete Scenario using DSL

The scenario-abstraction approach is illustrated in Figure 17. This process can be applied when detailed concrete use cases are available as part of the system requirements to validate the models. The process is very similar to the scenario-refinement approach (Figure 6), differing only in the artefacts used/produced and the method of constructing scenarios (step 3). There is an additional ‘given’ input; Concrete use cases, which is used to design the DSL (step 2) and construct scenarios (step 3).

– **Refinement Plan** Similar to the scenario-refinement approach, the first step is building a refinement plan from the system requirements to guide the modelling process.

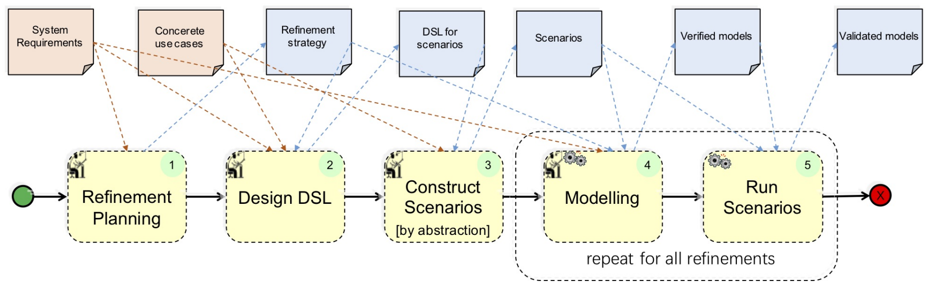


Fig. 17: Scenario-abstraction process

- **DSL** Considering the concrete use cases and any data refinement in the refinement plan, a DSL is designed as described in Section 5.
- **Scenarios** From the concrete use cases, scenarios are extracted for each refinement level in accordance with the refinement plan. The refined scenarios of the most concrete refinement level should be the same as the concrete use cases.
- **Modelling and Running Scenarios** Starting from the abstract level and for each refinement, a model is constructed in accordance with the refinement plan and guided by the extracted scenarios. As with the scenario-refinement approach, proof obligations may identify problems in the model but judgement should be used to decide whether it is more efficient to continue with validation before returning to discharge proofs manually. Validation of each model is done by running the scenarios using the model checker before moving to the next refinement. If any step leads to changes in the refinement plan, the extracted scenarios must be updated accordingly. However, the refined scenarios of the last refinement level must always be similar to the concrete use cases.

When extracting scenarios, a state that has been checked at a particular refinement level does not need to be checked at subsequent levels because the proof of refinement ensures 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. However, in some cases it is important to add extra checks using `Then` clause to check for negative scenarios. These scenarios can be part of the concrete use cases and are particularly important when it is not clear why in some cases a certain event is not enabled or when a state change can result in disabling a certain event.

In the rest of this section, we explain how the specification scenario is abstracted at the different levels 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. Figure 18 compares the abstract scenario with the concrete scenario, where the highlighted steps in Figure 18(b) corresponds to the train position on VSS, which are used in the abstract scenario (Figure 18(a)). At this stage, we have not yet introduced the concept of TTD, hence the `When` clause omits the TTD prefix and only defines the position of the train in terms of VSS. Since the concrete scenario concentrates on the internal state of the VBD controller, its `Then` clauses are not relevant to the abstract scenario. Instead we add new `Then` clauses that check the train's position in terms of VSS.

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. This scenario inherits the `When` clauses from the abstract scenario Figure 18(a), but with the addition of TTD prefixes since our model now incorporates TTD status. We then select `When` clauses related to radio connection and TTD state from the concrete scenario as highlighted in Figure 19(b). The `Then` clauses from the abstract scenario are omitted since they are guaranteed by refinement. Instead, we add new `Then` clauses to check train connection and TTD state after the `When` clauses that should affect this (Figure 19(a)).

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 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 (Figure 20).

Introduce position reports, VSS availability, integrity and distinguish between FS and OS MA. In this refinement, 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 (Figure 21).

Introduce timers. This refinement introduces propagation timers that expand the unavailable area of VSS in case a non-communicative train moves (Figure 22).

```
Given Train1 stood at VSS11
When Train1 enters VSS12
  Then Train1 stood at VSS11,VSS12
When Train1 leaves VSS11
  Then Train1 stood at VSS12
When Train1 enters VSS21
  Then Train1 stood at VSS12,VSS21
When Train1 leaves VSS12
  Then Train1 stood at VSS21
When Train1 enters VSS22
  Then Train1 stood at VSS21,VSS22
When Train1 leaves VSS21
  Then Train1 stood at VSS22
```

(a) Movement on VSS

```
Given Train1 stood at TTD10.VSS11
And Train1 disconnected
And TTD10.VSS11 is UNKNOWN
And TTD10.VSS12 is UNKNOWN
And TTD10 is OCCUPIED
And TTD20 is FREE
When Train1 starts mission and Train1
  connects
When Train1 reports position as integral
Then TTD10.VSS11 is AMBIGUOUS
When Train1 receives OSMA until TTD10.VSS12
When Train1 enters TTD10.VSS12
When Train1 leaves TTD10.VSS11
When Train1 reports position as integral
Then TTD10.VSS12 is AMBIGUOUS
And TTD10.VSS11 is UNKNOWN
When Train1 receives FSMA until TTD20.VSS22
When Train1 enters TTD20.VSS21
When Train1 leaves TTD10.VSS12
When Train1 reports position as integral
Then TTD10.VSS11 is FREE
And TTD10.VSS12 is FREE
And TTD20.VSS21 is OCCUPIED
And TTD10 is OCCUPIED
And TTD20 is FREE
When Train1 enters TTD20.VSS22
When Train1 leaves TTD20.VSS21
When Train1 reports position as integral
Then TTD20.VSS21 is FREE
And TTD20.VSS22 is OCCUPIED
When Train1 disconnects and Train1 ends
  mission
Then TTD20.VSS22 is UNKNOWN
Then TTD20.VSS22.
  disconnect_propagation_timer starts
When TTD20.VSS22.
  disconnect_propagation_timer expires
Then TTD20.VSS21 is UNKNOWN
And TTD20.VSS23 is UNKNOWN
```

(b) Concrete Scenario using DSL

Fig. 18: Deriving abstract scenario: train movement

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.

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 5.3.

```
Given Train1 stood at TTD10.VSS11
  And Train1 is disconnected
  And TTD10 is OCCUPIED
  And TTD20 is FREE
When Train1 connects
  Then Train1 connected
When Train1 enters TTD10.VSS12
When Train1 leaves TTD10.VSS11
When Train1 enters TTD10.VSS21
  Then TTD20 is OCCUPIED
When Train1 leaves TTD10.VSS12
  Then TTD10 is FREE
When Train1 enters TTD10.VSS22
When Train1 leaves TTD10.VSS21
When Train1 disconnects
  Then Train1 disconnected
```

(a) Radio connection

```
Given Train1 stood at TTD10.VSS11
And Train1 disconnected
And TTD10.VSS11 is UNKNOWN
And TTD10.VSS12 is UNKNOWN
And TTD10 is OCCUPIED
And TTD20 is FREE
When Train1 starts mission and Train1 connects
When Train1 reports position as integral
Then TTD10.VSS11 is AMBIGUOUS
When Train1 receives OSMA until TTD10.VSS12
When Train1 enters TTD10.VSS12
When Train1 leaves TTD10.VSS11
When Train1 reports position as integral
Then TTD10.VSS12 is AMBIGUOUS
And TTD10.VSS11 is UNKNOWN
When Train1 receives FSMA until TTD20.VSS22
When Train1 enters TTD20.VSS21
When Train1 leaves TTD10.VSS12
When Train1 reports position as integral
Then TTD10.VSS11 is FREE
And TTD10.VSS12 is FREE
And TTD20.VSS21 is OCCUPIED
And TTD10 is OCCUPIED
And TTD20 is FREE
When Train1 enters TTD20.VSS22
When Train1 leaves TTD20.VSS21
When Train1 reports position as integral
Then TTD20.VSS21 is FREE
And TTD20.VSS22 is OCCUPIED
When Train1 disconnects and Train1 ends mission
Then TTD20.VSS22 is UNKNOWN
Then TTD20.VSS22.disconnect_propagation_timer
     starts
When TTD20.VSS22.disconnect_propagation_timer
     expires
Then TTD20.VSS21 is UNKNOWN
And TTD20.VSS23 is UNKNOWN
```

(b) Concrete Scenario using DSL

Fig. 19: Deriving abstract scenario: train communication and TTD

8 Future Work

In future work we will continue to develop scenarios for the Tokeneer case study and investigate tool automation of the scenarios 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.

We use two tools to execute scenarios; the Scenario Checker for manual validation and Cucumber for Event-B for automated regression validation. It is

```
Given Train1 stood at TTD10.VSS11
And Train1 disconnected
And TTD10 is OCCUPIED
And TTD20 is FREE
When Train1 starts mission and Train1 connects
Then Train1 in mission
When Train1 receives MA until TTD10.VSS12
Then Train1 has MA until TTD10.VSS12
When Train1 enters TTD10.VSS12
When Train1 leaves TTD10.VSS11
When Train1 receives MA until TTD20.VSS22
Then Train1 has MA until TTD20.VSS12
When Train1 enters TTD20.VSS21
When Train1 leaves TTD10.VSS12
When Train1 enters TTD20.VSS22
When Train1 leaves TTD20.VSS21
When Train1 disconnects and Train1 ends mission
Then Train1 no mission
```

Fig. 20: Missions and generic MA

```
Given Train1 stood at TTD10.VSS11
And Train1 disconnected
And TTD10.VSS11 is UNAVAILABLE
And TTD10.VSS12 is UNAVAILABLE
And TTD10 is OCCUPIED
And TTD20 is FREE
When Train1 starts mission and Train1 connects
When Train1 reports position as integral
When Train1 receives OSMA until TTD10.VSS12
Then Train1 has OSMA until TTD10.VSS12
When Train1 enters TTD10.VSS12
When Train1 leaves TTD10.VSS11
When Train1 receives FSMA until TTD20.VSS22
Then Train1 has FSMA until TTD20.VSS22
When Train1 enters TTD20.VSS21
When Train1 leaves TTD10.VSS12
When Train1 reports position as integral
Then TTD10.VSS11 is AVAILABLE
And TTD10.VSS12 is AVAILABLE
And TTD10.VSS21 is UNAVAILABLE
When Train1 enters TTD20.VSS22
When Train1 leaves TTD20.VSS21
When Train1 reports position as integral
Then TTD10.VSS21 is AVAILABLE
And TTD10.VSS22 is UNAVAILABLE
When Train1 disconnects and Train1 ends mission
```

Fig. 21: Position reports, VSS avalability and integrity

desirable to have a common persistence in both tools using the DSL scenarios syntax. Figure 23 shows a proposed architecture for the tools. The Scenario Checker and Cucumber for Event-B share a common syntax for scenarios based on Event-B (i.e. events, variables etc. rather than DSL concepts). A generic con-

```
Given Train1 stood at TTD10.VSS11
And Train1 disconnected
And TTD10.VSS11 is UNAVAILABLE
And TTD10.VSS12 is UNAVAILABLE
And TTD10 is OCCUPIED
And TTD20 is FREE
When Train1 starts mission and Train1 connects
When Train1 reports position as integral
When Train1 receives OSMA until TTD10.VSS12
When Train1 enters TTD10.VSS12
When Train1 leaves TTD10.VSS11
When Train1 receives FSMA until TTD20.VSS22
When Train1 enters TTD20.VSS21
When Train1 leaves TTD10.VSS12
When Train1 reports position as integral
When Train1 enters TTD20.VSS22
When Train1 leaves TTD20.VSS21
When Train1 reports position as integral
When Train1 disconnects
When Train1 ends mission
Then TTD20.VSS22.disconnect_propagation_timer starts
When TTD20.VSS22.disconnect_propagation_timer expires
Then TTD20.VSS21 is UNAVAILABLE
And TTD20.VSS23 is UNAVAILABLE
```

Fig. 22: Timers

version tool loads/saves scenarios in a particular DSL, interpreting the mapping based on the DSL definition.

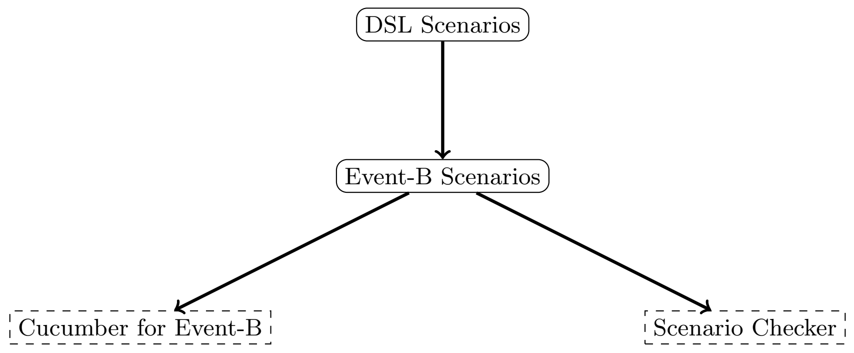


Fig. 23: DSL for Automatic and Interactive Validation

We plan to develop the Scenario Checker so that it becomes the user interface for running both the automatic and manual validation. We will investigate the potential for using the ProB to help discover refined scenarios as suggested in Section 6.1. ProB can be used to find a trace from the current step to a state where the next step in the abstract scenario is enabled, however, by default it returns only the first trace found.

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 adopt 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. Scenarios are a reformulation of the specification and, no matter what format they are expressed in, errors may be introduced. However, errors are equally likely to exist in the original specification. We have found that scenarios aid detection of specification errors by allowing validation of the behaviour by domain experts. If errors are introduced into the scenarios these will be discovered when they are used to animate the model.

An important step is to make the scenarios more precise so that they are clear and unambiguous while remaining easily understood by all stakeholders. To achieve this, we have suggested deriving a scenario DSL from the particular specification in question, prior to commencing the formal modelling. Scenarios that illustrate the desired behaviour embodied by the specification, may then be expressed in a clear, precise and concise way. 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.

When scenarios are not provided as part of the requirements they can be constructed to match the planned model refinements. In this case the scenarios are developed top down from an abstract version introducing the details in refined versions that match the model refinements and using the refined scenarios to explore the veracity of the model. Conversely, concrete use-case scenarios are sometimes provided to illustrate the meaning of a specification. 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 scenario-refinement and scenario-abstraction techniques both use the planned event and data refinement of the model to make corresponding refinements or abstractions to scenarios.

Although there is some useful previous work in this area, we bring together and discuss the different approaches available and provide tool support towards a ‘continuous integration’ style validation process. We hope this will help practitioners cope with the iterative nature of formal modelling as well as bridging the semantic gap between domain knowledge and formal precision.

Acknowledgements

This work was supported, in part, by the HICLASS project, funded by the Aerospace Technology Institute and Innovate UK, as project number 113213.

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. AdaCore. *Tokeneer case study by Praxis.* `https://www.adacore.com/tokeneer`. Accessed 31/12/2019.
4. Paolo Arcaini and Elvinia Riccobene. Automatic refinement of ASM abstract test cases. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2019, Xi'an, China, April 22-23, 2019*, pages 1–10, 2019.
5. Jean-Paul Bodeveix, Mamoun Filali, Julia L. Lawall, and Gilles Muller. Formal methods meet domain specific languages. In *Integrated Formal Methods, 5th International Conference, IFM 2005, Eindhoven, The Netherlands, November 29 - December 2, 2005, Proceedings*, pages 187–206, 2005.
6. M. Butler, D. Dghaym, T. S. Hoang, T. Omitola, C. Snook, A. Fellner, R. Schlick, T. Tarrach, T. Fischer, and P. Tummeltshammer. Behaviour-driven formal model development of the etcs hybrid level 3. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 97–106, 2019.
7. Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A scenario-based validation language for asms. In *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, pages 71–84, 2008.
8. John M. Carroll. Five reasons for scenario-based design. *Interacting with Computers*, 13(1):43–60, 2000.
9. Jacob L Cybulski. The formal and the informal in requirements engineering. Technical report, Technical Report 96/7, Department of Information Systems, The University of Melbourne, 1996.
10. 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.
11. Tomas Fischer. Cucumber for Event-B and iUML-B. `https://github.com/tofische/cucumber-event-b`, 2018.
12. Tomas Fischer and Dana Dghyam. Formal model validation through acceptance tests. In Simon Collart-Dutilleul, Thierry Lecomte, and Alexander Romanovsky, editors, *RSSRail 2019: Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, volume 11495 of *LNCS*, pages 159–169. Springer, 2019.
13. EEIG ERTMS Users Group. Hybrid ERTMS/ETCS Level 3:Principles, July 2017. Ref. 16E042 Version 1A.

14. Alexei Iliasov. Use case scenarios as verification conditions: Event-b/flow approach. In Elena Troubitsyna, editor, *Software Engineering for Resilient Systems - Third International Workshop, SERENE 2011, Geneva, Switzerland, September 29-30, 2011. Proceedings*, volume 6968 of *Lecture Notes in Computer Science*, pages 9–23. Springer, 2011.
15. Phillip James and Markus Roggenbach. Encapsulating formal methods within domain specific languages: A solution for verifying railway scheme plans. *CoRR*, abs/1403.3034, 2014.
16. Qaisar A. Malik, Johan Lilius, and Linas Laibinis. Model-Based Testing Using Scenarios and Event-B Refinements. In Michael Butler, Cliff Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *Methods, Models and Tools for Fault Tolerance*, pages 177–195, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
17. Mar Yah Said, Michael Butler, and Colin Snook. A method of refinement in UML-B. *Softw. Syst. Model.*, 14(4):1557–1580, October 2015.
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 Snook, Thai Son Hoang, Dana Dghaym, and Michael Butler. Domain-specific scenarios for refinement-based methods. In Christian Attiogbé, Flavio Ferrarotti, and Sofian Maabout, editors, *New Trends in Model and Data Engineering*, pages 18–31, Cham, 2019. Springer International Publishing.
21. 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, *ICFEM2018*, volume 11232 of *LNCS*, pages 21–36. Springer, 2018.
22. Stefan Sobernig, Bernhard Hoisl, and Mark Strembeck. Requirements-driven testing of domain-specific core language models using scenarios. In *2013 13th International Conference on Quality Software, Najing, China, July 29-30, 2013*, pages 163–172, 2013.
23. Matt Wynne and Aslak Hellesøy. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Programmers, LLC, 2012.