The original publication is available at
http://doi.org/10.1007/978-3-030-02450-5_2
In ICFEM 2018: Proceedings of the 20th International Conference on Formal Engineering Methods © Springer 2018

Behaviour-driven formal model development

Colin Snook[1]✉https://orcid.org/0000-0002-0210-0983, Thai Son Hoang[1]https://orcid.org/0000-0003-4095-0732, Dana Dghyam[1]https://orcid.org/0000-0002-2196-2749, Michael Butler[1]https://orcid.org/0000-0003-4642-5373, Tomas Fischer[2], Rupert Schlick[3]https://orcid.org/0000-0002-5644-1679, and Keming Wang[4]

[1] ECS, University of Southampton, Southampton, U.K.,
{cfs, t.s.hoang, D.Dghaym, mjb}@ecs.soton.ac.uk
[2] Thales Austria GmbH, Vienna, Austria,
tomas.fischer@thalesgroup.com
[3] AIT Austrian Institute of Technology GmbH, Vienna, Austria,
Rupert.Schlick@ait.ac.at
[4] Southwest Jiaotong University, Chengdu, China
kmwang@swjtu.edu.cn

Abstract. Formal systems modelling offers a rigorous system-level analysis resulting in a precise and reliable specification. However, some issues remain: Modellers need to understand the requirements in order to formulate the models, formal verification may focus on safety properties rather than temporal behaviour, domain experts need to validate the final models to ensure they fit the needs of stakeholders. In this paper we discuss how the principles of Behaviour-Driven Development (BDD) can be applied to formal systems modelling and validation. We propose a process where manually authored scenarios are used initially to support the requirements and help the modeller. The same scenarios are used to verify behavioural properties of the model. The model is then mutated to automatically generate scenarios that have a more complete coverage than the manual ones. These automatically generated scenarios are used to animate the model in a final acceptance stage. For this acceptance stage, it is important that a domain expert decides whether or not the behaviour is useful.

Keywords: Formal modelling; Scenarios; Mutation testing; Acceptance testing

1 Introduction

For highly dependable systems, formal modelling offers a rigorous system-level analysis to ensure that the specification is consistent with important properties

All data supporting this study are openly available from the University of Southampton repository at http://doi.org/10.5258/SOTON/D0604.

such as safety and security. Using theorem provers, such properties can be proven to hold generically without instantiation and testing. However, modellers need to understand the requirements in order to formulate correct and useful models. The human centric processes of understanding a natural language or semi-formal requirements document and representing it in mathematical abstraction is subjective and intellectual, leading to misinterpretation. Formal verification may then focus on safety properties rather than desired behaviour which is more difficult to verify as a proof obligation. Even if these difficulties are averted, the requirements may not represent the customer's needs. Domain experts need to validate the final models to show that they capture the informally specified customer requirements and ensure they fit the needs of stakeholders.

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

Behaviour-Driven Development (BDD) [13,16] is a software development process based on writing precise semi-formal scenarios as a behavioural specification and using them as acceptance tests. In this paper we discuss how the principles of BDD can be applied to formal systems modelling and validation. We propose a process where manually authored scenarios are used initially to support the requirements and help the modeller. The same scenarios are used to verify behavioural properties of the model. However, the manually written tests may have limited coverage. To address this, the model is mutated to automatically generate further scenarios that have a more complete coverage than the manual ones. The additional scenarios should be accepted or rejected by domain experts to ensure they, and hence the model, represent the desired behaviour. These automatically generated scenarios are used to animate the model in a final acceptance stage. For this acceptance stage, it is important that a domain expert decides whether or not the behaviour is desirable.

Customer requirements are typically based on a domain model, which is often expressed in terms of entities with attributes and relationships. State-machines and activity diagrams can be used to describe the behaviour. On the other hand, a formal model (such as Event-B) is based on set theory and predicate logic [1]. In a creative process, the modelling engineer uses ingenuity to translate the domain model into appropriate formal structures. The mismatch between the semi-formal models understood by the domain experts and the mathematical notations used for formal modelling leads to a conflict. The acceptance tests need to be expressed in terms of the formal model, but they also need to be understood by the domain experts who are not familiar with the formal notations. It would be more desirable to express the acceptance tests in terms of the domain model so that domain experts can easily create and validate them.

iUML-B [14,17,18] provides a UML-like diagrammatic modelling notation, including class diagrams and state-machines, with automatic generation of Event-B formal models. Hence, iUML-B is a formal notation which is intuitive to write and understand and is much closer to the domain model.

Gherkin [20, Chapter 3] is a structured language for describing scenarios and expected behaviour in a readable but executable form. In this paper we show how Gherkin supported by the *Cucumber* tool, can be used to encode and execute acceptance tests for validating Event-B and iUML-B formal models. This helps domain experts by allowing them to define acceptance tests without requiring expertise in formal modelling. It also helps the formal experts by providing means to systematically validate formal models via input from domain experts.

The remainder of the paper is structured as follows. In Section 2 we introduce the "Lift" examples used throughout the paper. In Section 3 we provide an overview of the Cucumber framework and Gherkin notation for executing scenarios, the formal methods Event-B and iUML-B that we use and MoMuT which we use as a scenario generation tool. In Section 4 we introduce our approach to behaviour-driven formal model development and then, in Section 5, demonstrate how to use Gherkin and Cucumber for testing formal models written in Event-B and iUML-B. Section 6 describes related work and section 7 concludes.

2 Running Examples

This section gives a brief overview of our running examples. The main running example in this paper is a single-shaft lift controller. In Section 5.2 , we extend this example to a multi-shaft lift controller to illustrate our contribution on linking Gherkin/Cucumber with iUML-B.

A Single-Shaft Lift. First we consider a single shaft lift operating between several floors, Fig. 1. The cabin has request buttons for each floor and each floor has an up and down request button. The cabin is moved up and down by winding, resp. unwinding, a motor. The cabin door may only open when the lift is not moving. The full requirements of the single-shaft lift are given in [4]. The cabin should only move to respond to requests and should only change direction when there are no requests ahead in its direction of travel. Any requests associated with the current floor are cleared when the door begins to open.

A Multi-Shaft Lift. This system manages multiple lifts with a single cabin in each shaft. The behaviour of the cabin motor and door is similar to the single-shaft lift. Similarly, the cabin floor requests are dealt with internally by each lift. The main difference is in the up/down requests at the floor levels. The up/down floor requests are assigned by a central controller to the nearest serving cabin.

The nearest cabin is determined by calculating the ‘figure of suitability’ of each lift, which depends on the direction of the lift, the direction of the call and the distance to the calling floor. Once a request is assigned to a lift, the cabin will serve the request similar to the single shaft example. The full requirements of the multi-shaft lift are given in [4].

3 Background and Technologies

In this section, we first review the Gherkin/Cucumber approach to BDD, followed by a short description of the Event-B method and its iUML-B diagrammatic notation.

3.1 Behaviour-Driven Development with Gherkin/Cucumber

The 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 [20] is one of the various frameworks supporting BDD [19].

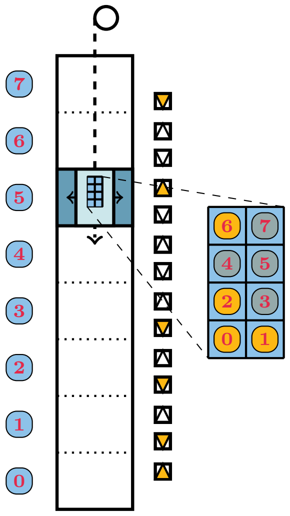

Fig. 1. A Lift System

Gherkin. Gherkin [20, 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 feature starts with some description, followed by a list of scenarios. The feature is often written as a story, e.g.,

“*As a* ≪role≫*I want* ≪feature≫*so that* ≪business value≫”.

Scenario. Each scenario represents one use case. There are no technical restrictions about the number of scenarios in a feature; yet they all should be related to the feature being described.

In the simplest case the scenario also contains the test data and thus represents an individual test case. It is however advantageous to separate the general requirement description from the concrete test cases and to describe a group of similar use cases at once. For this purpose, a scenario outline with a placeholder for the particular test data specified separately as a list of examples can be used. In the following, we focus on different scenario steps.

Steps. Every 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.

Cucumber. Cucumber is a framework for executing acceptance tests written in Gherkin language and provides Gherkin language parser, test automation as well as report generation. In order to make such test cases automatically executable, the user must supply the actual step definitions providing the gluing code, which implements the interaction with the System Under Test (SUT). The steps shall be written in a generic way, i.e. serving multiple features. This keeps the number of step definitions much smaller than the number of tests. It is an antipattern to supply feature-coupled step definitions which cannot be re-used across features or scenarios.

Compound steps may encapsulate complex interaction with a system caused by a single domain activity, thus decoupling the features from the technical interfaces of the SUT. This defines a new domain-related testing language, which may simplify the feature description. The description of the business functionality is, however, still contained in the features.

An example of a scenario for the single-shaft lift system is shown in Listing 1.

3.2 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* $\mathsf{I}(\mathsf{v})$ 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, $\mathsf{G}(\mathsf{t}, \mathsf{v})$ is the guard of the event, and $\mathsf{v} := \mathsf{E}(\mathsf{t}, \mathsf{v})$ is the action of the event.

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

```
Scenario: Press a DOWN button

Given can press DOWN button at floor "2"
When press DOWN button at floor "2"
Then DOWN button at floor "2" is lit
And can wind the lift motor
And cannot open door

When motor starts winding
Then lift can move up
And cannot open door

When lift moves up
Then floor is "1"
And lift can move up
```

Listing 1. A test scenario for single-shaft lift

Actions in Event-B are, in the most general cases, non-deterministic [7], e.g., of the form $v :\in E(v)$ (v is assigned any element from the set $E(v)$) or $v :| P(v,v')$ (v is assigned any value satisfying the before-after predicate $P(v,v')$). A special event called INITIALISATION without parameters and guards is used to put the system into the initial state.

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

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

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

3.3 MoMuT

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

In contrast to other model based testing tools, the generated test cases do not target structural coverage of the model, but target exposing artificial faults systematically injected into the model. These faults are representatives of potential faults in the implementation; a test finding them in the model can be assumed to find its direct counterpart as well as similar, not only identical problems in the implementation [6].

As input models, MoMuT accepts Object Oriented Action Systems (OOAS) [9], an object oriented extension of Back's Action systems [3]. The underlying concepts of Action systems and Event-B are both closely related to Dijkstra's guarded command language [5]. For a subset of UML, for some Domain Specific Languages (DSLs) and for a subset of Event-B, transformations into OOAS are available.

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

3.4 iUML-B

iUML-B [14,17,18], an extension of the Rodin Platform, provides a 'UML like' diagrammatic modelling notation for Event-B in the form of class-diagrams and state-machines. The diagrammatic elements are contained within an Event-B model and generate or contribute to parts of it. The iUML-B makes the formal models more visual and thus easier to comprehend. We omit the description of state-machines and focus on class-diagrams, which are used in the example in Section 5.2.

Class diagrams provide a way to visually model data relationships. Classes, attributes and associations are linked to Event-B data elements (carrier sets, constants, or variables) and generate constraints on those elements. Methods elaborate Event-B events and contribute additional parameter representing the class instance.

4 Behaviour-Driven Formal Model Development

In this section, we present our approach for behaviour-driven formal model development. We assume that a natural language description of the requirements is available and this is supported by a number of manually written scenarios. The process, shown in Figure 2, consists of the following steps.

1. In the *modelling* step, the model is produced from the *requirements* and the *manually written scenarios*. The output of the modelling step is a *safe model*, in the sense that it is fully proven to be consistent with respect to its invariants. (We use 'safe' in a wide sense to include any important properties). In this paper, we use Event-B/iUML-B as our modelling method.
2. The safe model is *behaviourally verified* against the manually written scenarios. The purpose is to verify that the safe model exhibits the behaviour specified in the requirements which cannot be expressed via invariants. The output of this step is a (safe and) *behaviourally verified model*. In this paper, we use Cucumber for Event-B/iUML-B (see Section 5) for verifying the behaviour of our model written in Event-B/iUMLB.

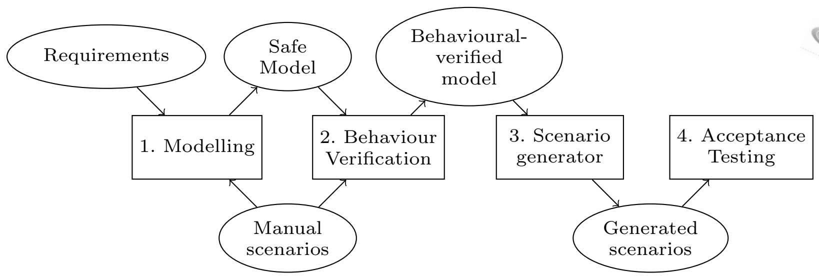

Fig. 2. A Behaviour-Driven Formal Model Development Method

3. The behaviourally-verified model is used as the input for a *scenario generator*, which automatically produces a collection of *generated scenarios*. In this paper, we use an Event-B-enabled version of MoMuT (see Section 3) as the scenario generator. The generated scenarios should be reviewed to ensure they represent desired behaviour. If the model still contains undesirable behaviour, that was not detected in the previous step, this will be reflected in the generated scenarios.
4. The generated scenarios are used for acceptance testing of the behaviourally verified model. Acceptance testing allows stakeholders to assess the usefulness of the model by watching its behaviour. We again use Cucumber for Event-B/iUML-B to automatically illustrate the generated scenarios to different stakeholders. The scenarios are in "natural language" and it is easy to see the correspondence between the scenarios and the requirements.

Our hypotheses about our approach are as follows.

H1 In the modelling step, scenarios help to improve the validity of the model.
H2 Scenarios are useful for verifying temporal properties.
H3 Generated scenarios are more complete than manually written scenarios.

In the following sections, we analyse the steps of the process in more detail with experiments to verify the above hypotheses.

4.1 Modelling

To validate Hypothesis **H1**, we perform an experiment using the single-shaft lift controller introduced in Section 2. The requirements of the system are given to two developers who are expert in Event-B modelling. To one developer, we also gave a set of desirable scenarios of the system. The full scenarios can be seen in [4]. The summary of the scenarios is as follows.

Scenario 1 User 1 enters the lift from Floor 0 and presses the button for Floor 2. User 2 presses the *up* button on Floor 1. The lift will go from Floor 0 to Floor 2, in between stop at Floor 1 to serve User 2's request.

Scenario 2 User 1 enters the lift from Floor 0 and presses the button for Floor 2. User 2 presses the *down* button on Floor 1. The lift will first go from Floor 0 to Floor 2, before changing the direction to go down to Floor 1 to serve User 2's request.

Scenario 3 User 1 enters the lift from Floor 0 and presses the button for Floor 2. User 2 presses the *down* button on Floor 1. The lift will first go from Floor 0 to Floor 2, before changing the direction to go down (still at Floor 2) to serve User 2's request.

Afterwards, we compare the models produced by the two developers in terms of their validity with respect to the requirements and the scenarios. The comparison is done by executing the scenarios on the models and reviewing their behaviour.

We did not find much difference in terms of valid behaviour between the two models. This may be due to tacit knowledge of the lift example. However, we found that the scenarios have some effect on the form of the models. The model developed with scenarios aligns more closely with the details presented in the scenario: The lift responses directly to the buttons pressed by the users. In the model developed without scenarios, an abstract notion of "requests" is introduced, which are eventually linked with the buttons. Having such a strong example of actual behaviour seems to *reduce the inclination to make abstractions*. On the one hand, the model without abstraction has less refinement steps and is more obviously valid since it directly correlates with the acceptance criteria. On the other hand, the model with abstraction has principles that can be adapted to different concrete implementations and hence may be more reusable. Scenarios help with validation of the models but may reduce their reusability. A possible mitigation is to develop "abstract" scenarios from the original concrete scenarios. We consider this as a direction for our future work.

4.2 Behaviour Verification

In this section, we describe our experiment to validate Hypothesis **H2**. The purpose of the behaviour verification step is to ensure that our safe model also satisfies behaviours which are specified using the scenarios. We use versions of the single-shaft lift model that has been seeded with several faults as follows.

1. (**Fault 1**) The lift is prevented from moving to the top floor. Event MovesUp's guard is changed from $\mathsf{floor} < \mathsf{TOP\_FLOOR}$ to $\mathsf{floor} < \mathsf{TOP\_FLOOR} - 1$.
2. (**Fault 2**) The up requests are not cleared after the door is open. Here the action to clear the up button for floor f, i.e., $\mathsf{up\_buttons} := \mathsf{up\_buttons} \setminus \{\mathsf{floor}\}$, is omitted in the faulty version of event UpButtonCleared.
3. (**Fault 3**) The down requests are ignored by the door, i.e., the door will not open if there is only a down request at a floor. Here, a guard of event DoorClosed2Half is changed from

$$\mathsf{direction} = \mathsf{DOWN} \Rightarrow \mathsf{floor} \in \mathsf{floor\_buttons} \cup \mathsf{down\_buttons}$$

to

$$\mathsf{direction} = \mathsf{DOWN} \Rightarrow \mathsf{floor} \in \mathsf{floor\_buttons}$$

These type of faults are typical in developing system models using Event-B and are *not detected* by verification using invariant proofs. In other words, the models with temporal faults are still fully proved to be consistent with their safety invariants.

In these experiments the manual scenarios found two of the seeded faults. Fault 2 is found by all scenarios, while Fault 3 is found by **Scenario 2** and **Scenario 3**. Since none of the manual scenarios get the lift cabin to the top floor, Fault 1 is not discovered. Nevertheless, our experiment confirms that the scenarios are useful for verifying behaviours of the system, which cannot be directly expressed and verified using invariants. In general, scenarios must also be verified and validated to ensure that they represent desirable behaviours of the system.

4.3 Scenario Generator

In this section, we verify Hypothesis **H3** by comparing the scenarios generated automatically by MoMuT with the manually written scenarios. We use MoMuT as our scenario generator on the model of the lift example. The generator explores a subset of the model's state space and checks where mutations, like exchanged operators or conditions set to a fixed value, cause the externally visible behaviour to differ from the original model. This information is used to build test scenarios that succeed on the original model, but fail on a model containing the mutation.

For the exploration, we tried three strategies: (a) random exploration, (b) exploration using rapidly expanding random trees (RRT) and (c) full exploration up to depth 12 (BFS12). The exploration depth for BFS12 was limited by the memory of the computer we used.

Table 1. Comparison of Scenario Sets

Scenario Set	Fault 1	Fault 2	Fault 3	Coverage	Steps
Manual	No	Yes	Yes	72 %	87
Random	Yes	Yes	Yes	63 %	305
RRT	Yes	Yes	Yes	67 %	204
BFS12	No	Yes	Yes	79 %	82

Table 1 shows, for each generation strategy, which of the manually seeded faults was detected, what percentage of the automatically generated model mutation faults were detected and the length in steps of the generated scenarios. A mutant is found when, during the exploration of the model, the modelling element (here the Event-B event) containing the mutant is executed. As a result, the (mutant) coverage criteria is a property of the scenario sets with respect to the formal model.

As can be seen in Table 1, the manual set already achieves a high mutation coverage of 72% of the 616 inserted mutations, and is only outperformed by the BFS12 scenarios, achieving higher coverage (79%) with even fewer steps. Nonetheless, both the manual set and BFS12 fail to catch our first seeded fault, because both do not try to go to the third floor. The scenarios from the two other strategies catch the first seeded fault, but perform less well regarding overall coverage number and coverage achieved in relation to steps needed.

Analysis of the generated scenarios shows that the different groups of scenarios do not subsume each other. Thus, putting all automatically generated scenarios together, an even higher mutation coverage score of 83% can be reached. Although the gap is smaller than expected, the experimental results support Hypothesis **H3**.

Since the overall size of the scenario sets is not too much bigger than the manual scenarios, manual review of the generated scenarios is feasible. Automated reduction of the tests or more optimised generation techniques would improve that even more. Longer random scenarios could increase the fault-finding capacity, but at the cost of review feasibility. The problem with random tests is not only the length of the scenarios. The more random a generated scenario is, the more tiresome it is to work through during acceptance testing, because there is no intention recognisable.

5 Scenario Automation for Event-B/iUML-B

In this section, we present our Cucumber step definitions for Event-B and iUML-B. Cucumber for Event-B/iUML-B allows us to execute the Gherkin scenario directly on the Event-B/iUML-B models.

5.1 Automation: Cucumber for Event-B

'Cucumber for Event-B' allows Cucumber to execute Gherkin scenarios on an Event-B model. It is a collection of step definitions which defines a traversal of the Event-B state space. Below we intersperse the Gherkin step definitions with comments to explain how to interpret them.

Given *machine with* "≪formula≫"
  // Setup constants with the given constraints and initialize the machine.
When *fire event* "≪name≫" *with* "≪formula≫"
  // Fire the given event with the given parameter constraints.
Then *event* "≪name≫" *with* "≪formula≫" *is enabled*
  // Check if the given event with the given parameter constraints is enabled.
Then *event* "≪name≫" *with* "≪formula≫" *is disabled*
  // Check if the given event with the given parameter constraints is disabled.
Then *formula* "≪formula≫" *is TRUE*
  // Check if the given formula evaluates to TRUE.
Then *formula* "≪formula≫" *is FALSE*
  // Check if the given formula evaluates to FALSE.

An essential property of acceptance tests is reproducibility. Therefore all step definitions check whether the specified event can be unambiguously chosen (using given parameters constraints). The user should make sure that the tested machine is deterministic and, if not, refine it further. Also abstract constants may lead to non-reproducible tests; however, they do not need to be specified by the model refinement, but can also be provided by the test case as test data.

The scenario to test the functionality of a single-shaft lift system in Listing 1 can be rewritten for the Event-B model as shown in Listing 2.

```
Scenario: Press the DOWN button

Given machine with "TopFloor = 3"
When fire event "DownButtonPresses" with "f = 2"
Then formula "2 : down_buttons" is TRUE
And event "MotorWinds" is enabled
And event "DoorClosed2Half" is disabled

When fire event "MotorWinds"
Then formula "motor = WINDING" is TRUE
And event "MovesUp" is enabled
And event "DoorClosed2Half" is disabled

When fire event "MovesUp"
Then formula "floor = 1" is TRUE
And event "MovesUp" is enabled
```

Listing 2. Test scenario using plain step definitions

Such an acceptance test is fairly straightforward in terms of syntax but is couched in terms of the relatively low-level formalism of Event-B. Domain engineers are often more used to higher-level modelling representations such as UML. In Section 5.2 we go further towards meeting the BDD approach which advocates minimising the language barriers between domain and system engineers.

5.2 Cucumber for iUML-B

Cucumber for iUML-B provides a Gherkin syntax based on the iUML-B diagrammatic modelling notation. iUML-B class diagrams and state-machines resemble the equivalent notations of UML and should feel more familiar for domain engineers. For the multi-shaft lift example, we have used iUML-B class diagrams to illustrate scenario testing of behaviour 'lifted' to a set of instances (i.e. a class). Although not shown here, Cucumber for iUML-B also supports scenario testing of state-machines including state-machines that are owned (i.e. contained) by a class in a class diagram.

Cucumber for iUML-B consists of iUML-B based step definitions which are translated into the corresponding underlying Event-B model elements for execution. Clearly, the translation of Cucumber for iUML-B scenarios must match the corresponding translation of the actual target model under test. Therefore Cucumber for iUML-B must access attributes of the iUML-B model in order to infer the proper Event-B events and variables and to derive implicit event parameters (e.g. 'self name' representing the class instance).

Cucumber for Class Diagrams. The following Gherkin syntax is be defined for validating class diagrams.

Given *class* "≪name≫:≪inst≫"
  // Preset the given class with the given instance.
When *call method* "≪name≫" *with* "≪formula≫"
  // Call the given class instance method.
Then *method* "≪name≫" *with* "≪formula≫" *is enabled*
  // Check if the given class instance method is enabled.
Then *method* "≪name≫" *with* "≪formula≫" *is disabled*
  // Check if the given class instance method is disabled.
Then *attribute* "≪attr≫" *is* "≪value≫"
  // Check if the given class instance attribute is equal to the given value.
Then *association* "≪assoc≫" *is* "≪value≫"
  // Check if the given class instance association is equal to the given value.

In general, class attributes and associations can be any binary relation (i.e., not necessarily functional), hence further checks can be defined accordingly.

Multi-shaft lift system in iUML-B class diagrams. Figure 3 represents the class diagram of the lift requests, before introducing the motor and door behaviour. Class Bldg_Lift is a constant representing the lift cabins in a building. Each lift has two attributes lift_status and lift_direction to indicate whether the lift is moving or not and the lift direction (up/down). Floors is a constant representing the different floors in a building.

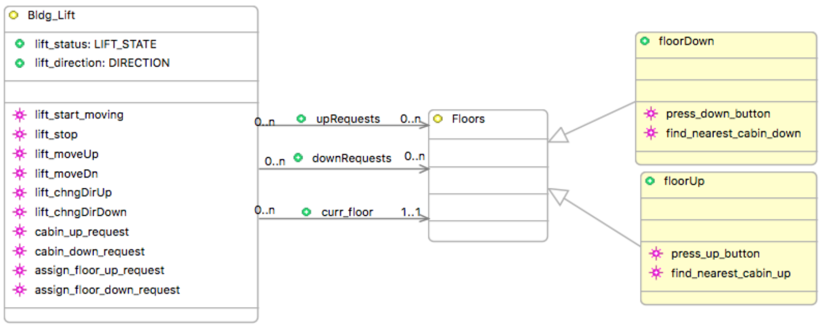

Fig. 3. iUML-B class diagram of the multi shaft lift: Requests

The associations upRequests and downRequests between the Bldg_Lift and Floors are variables that represent the floors to be served by the lift, and whether they are above or below the current position of the lift, while curr_floor represents the current floor position of the lift. floorUp and floorDown are variable sets of type Floors, that respectively represent floor up and down requests, these requests are generated by the events press_up_button and press_down_button. At this stage the floor requests are not assigned to a specific lift, once the controller finds the nearest serving lift (find_nearest_cabin_up,find_nearest_cabin_down), these requests will be assigned to the nearest lift in the Bldg_Lift events assign_floor_up_request and assign_floor_down_request. The Bldg_Lift has other local events e.g. cabin_up_request, lift_move_up etc.

The Scenario of Listing 3 tests the action of requesting a floor from within a cabin of the multi-shaft lift system modelled in Figure 3. Note that we use *with* "≪formula≫" to instantiate the additional parameter f to specify the requested floor for the given building lift L1.

```
Scenario: Request cabin floor

Given class "Bldg_Lift:L1"
Then method "cabin_up_request" with "f = 1" is enabled
And method "cabin_down_request" with "f = 1" is disabled
And attribute "lift_status" is "STATIONARY"
And attribute "lift_direction" is "UP"

When call method "cabin_up_request" with "f = 1"
Then association "upRequests" is "{1}"
And method "lift_start_moving" is enabled

When call method "lift_start_moving"
Then attribute "lift_status" is "MOVING"
```

Listing 3. Test scenario for iUML-B class diagram

6 Related Work

Our approach is inspired by the behaviour-driven development methods [13,16] of agile methods. Siqueira, deSousa and Silva [15] also propose using BDD with Event-B. However, they use Event-B to support the BDD process by providing it with better analyses whereas we retain focus on formal modelling using 'BDD-like' techniques to improve our model development process. The concept of acceptance testing of a formal model is perhaps unusual, however it builds on the idea of model validation via animation which has been supported for some time particularly in Event-B, with tools such as ProB [11] and BMotion Studio [10,12]. Acceptance testing is a more specific use of such validation tools where the goal is not only to validate the model but to allow the end-user or similar stakeholder to assess and accept the model as suitable for their needs.

7 Conclusion

We have developed an approach to formal modelling based on ideas from Behaviour Driven Development. We use scenarios to drive the formal model construction, verification and acceptance. We have shown how to enhance Cucumber in order to apply the acceptance tests written in the Gherkin language to the Event-B formal model and also to a model formulated using iUML-B notation. For efficient coverage we use a model-mutation based test case generator to generate scenarios for acceptance testing. Our experiments support the ideas but were somewhat neutral in the case of H1: ‘scenarios help to improve the validity of the model’. Further experiments will be carried out in this area on larger and less familiar applications where tacit knowledge is less likely to confound results. For example a different modeller could develop a new feature to assess whether scenarios help to identify the scope of impact of the change in a situation where the style of the overall model is already fixed. We would also like to explore the relationship between scenario testing and verification of temporal properties such as ‘does the lift eventually reach a requested floor’. This could be explored in relation to ‘lifted’ behaviours such as found in the multi-shaft lift where we might want to examine local liveness properties of classes. The test case generation, while having greater coverage than the manually written scenarios, did miss part of the seeded bugs depending on the selected search strategy. We believe this can be addressed by tuning the MoMuT tools and will carry out further work and experiments in this area. Our prototype tool can be found under https://github.com/tofische/cucumber-event-b. Further work is needed to develop the methods and tools to support the use of Cucumber for iUML-B. Our next applications will be in the railway domain on the Hybrid ERTMS/ETCS Level 3 [8] and in the avionics domain on an aircraft turn-around security authentication system, which are real industrial applications.

Acknowledgements

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

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

We also thank Thorsten Tarrach (Austrian Institute of Technology, Vienna, Austria) for his assistance with MoMuT.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)

2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An open toolset for modelling and reasoning in Event-B. Software Tools for Technology Transfer 12(6), 447–466 (Nov 2010)
3. Back, R., Sere, K.: Stepwise refinement of action systems. In: International Conference on Mathematics of Program Construction. pp. 115–138. Springer (1989)
4. Dghyam, D., Hoang, T.S., Snook, C.: Requirements document, scenarios, and models for lift examples (May 2018), http://doi.org/10.5258/SOTON/D0604
5. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
6. Fellner, A., Krenn, W., Schlick, R., Tarrach, T., Weissenbacher, G.: Model-based, mutation-driven test case generation via heuristic-guided branching search. In: Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design. pp. 56–66. ACM (2017)
7. Hoang, T.S.: An introduction to the Event-B modelling method. In: Industrial Deployment of System Engineering Methods, pp. 211–236. Springer-Verlag (2013)
8. Hoang, T., Butler, M., Reichl, K.: The hybrid ERTMS/ETCS level 3 case study. In: Butler, M., Raschke, A., Hoang, T., Reichl, K. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 251–261. Springer International Publishing (2018)
9. Krenn, W., Schlick, R., Aichernig, B.K.: Mapping UML to labeled transition systems for test-case generation. In: Formal Methods for Components and Objects. pp. 186–207. Springer (2010)
10. Ladenberger, L., Bendisposto, J., Leuschel, M.: Visualising Event-B models with B-Motion Studio. In: Proceedings of FMICS 2009. Lecture Notes in Computer Science, vol. 5825, pp. 202–204. Springer (2009)
11. Leuschel, M., Butler, M.: ProB: An automated analysis toolset for the B method. Software Tools for Technology Transfer (STTT) 10(2), 185–203 (2008)
12. Lukas Ladenberger: BMotion Studio for ProB project website. http://stups.hhu.de/ProB/w/BMotion_Studio (Jan 2016)
13. North, D.: Introducing BDD. Better Software Magazine (Mar 2006)
14. Said, M.Y., Butler, M., Snook, C.: A method of refinement in UML-B. Softw. Syst. Model. 14(4), 1557–1580 (Oct 2015), http://dx.doi.org/10.1007/s10270-013-0391-z
15. Siqueira, F.L., de Sousa, T.C., Silva, P.S.M.: Using BDD and SBVR to refine business goals into an Event-B model: A research idea. In: 2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE). pp. 31–36 (May 2017)
16. Smart, J.F.: BDD in Action: Behavior-Driven Development for the Whole Software Life cycle. Manning Publications Company (2014)
17. Snook, C.: iUML-B statemachines. In: Proceedings of the Rodin Workshop 2014. pp. 29–30. Toulouse, France (2014), http://eprints.soton.ac.uk/365301/
18. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. 15(1), 92–122 (Jan 2006), http://doi.acm.org/10.1145/1125808.1125811
19. Solis, C., Wang, X.: A study of the characteristics of behaviour driven development. In: 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications. pp. 383–387 (Aug 2011)
20. Wynne, M., Hellesøy, A.: The Cucumber Book: Behaviour-Driven Development for Testers and Developers. Pragmatic Programmers, LLC (2012)