Formal Model Validation through Acceptance Tests

Tomas Fischer[1] and Dana Dghyam[2]

[1] Thales Austria GmbH, Vienna, Austria,
tomas.fischer@thalesgroup.com
[2] ECS, University of Southampton, Southampton, U.K.,
dd4g12@ecs.soton.ac.uk

Abstract. When formal systems modelling is used as part of the development process, modellers need to understand the requirements in order to create appropriate models, and domain experts need to validate the final models to ensure they fit the needs of stakeholders. A suitable mechanism for such a validation are acceptance tests.
In this paper we discuss how the principles of Behaviour-Driven Development (BDD) can be applied to i) formal modelling and ii) validation of behaviour specifications, thus coupling those two tasks. We show how to close the gap between the informal domain specification and the formal model, thus enabling the domain expert to write acceptance tests in a high-level language matching the formal specification.
We analyse the applicability of this approach by providing the Gherkin scenarios for an Event-B/iUML-B formal model of a 'fixed virtual block' approach to train movement control, developed according to the Hybrid ERTMS/ETCS Level 3 principles specified by the EEIG ERTMS Users Group and presented as a case study on the 6. International ABZ Conference 2018.

Keywords: Formal Methods; Validation; Acceptance Tests; Event-B; iUML-B; Gherkin; Cucumber

1 Introduction

A fully proven formal model is still pointless if it does not represent the customer's needs. Therefore formal models must be thoroughly validated in order to show that they capture useful functionality. However, today's formal methods tools offer limited support for validation and it remains essentially a manual task, e.g. expert review, which is tedious, time consuming and error prone.

One widely-used and reliable validation method is acceptance testing, which, assuming adequate coverage, can provide assurance that a system (in our case embodied by the formal model) does indeed represent the informal customer requirements. Acceptance tests describe a sequence of stimulation steps involving concrete data examples to test the functional responses of the system. However, acceptance tests can also be viewed as a collection of user 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 over evolutions of the system. As such, the acceptance tests may be seen as the single reference or *source of truth*.

Behavior-driven development (BDD) methodology combines the general techniques and principles of test-driven development with ideas from domain-driven design and object oriented methods. It advocates that tests should be written first, describing desired functionality. Then the actual functionality should be implemented (or as in our case, a model should be created) to match the formulated requirements.

The remainder of the paper is structured as follows.

In Section 2 we give a brief overview of an Event-B/iUML-B formal model of a 'fixed virtual block' approach to train movement control.

In Section 3 we provide a short description of the Gherkin notation and Cucumber framework and demonstrate the validation of the presented Event-B/ iUML-B models using Gherkin acceptance tests. In the same section we analyse discovered problems and challenges and suggest further improvements.

In Section 4 we summarise the benefits of the approach for validating formal models and outline how the proposed method and tools will integrate into the formal modelling process being developed in the ENABLE-S3 project.

2 Hybrid ERTMS/ETCS Level 3

In this paper we use the Event-B model of a hybrid ERTMS/ETCS Level 3 (HL3) [4] presented in [3]. HL3 is a 'fixed virtual block' approach to train movement, where the trackside train detection (TTD) section is divided into a fixed number of virtual sections (VSS). A train movement controller called the Radio Block Centre (RBC) manages the Movement Authority (MA) granted to each train in mission. This granted MA is the permission for a train to move safely to a specific location avoiding train collisions. However, in order for the RBC to grant a MA it needs to know which sections are free. The status of the virtual sections is calculated by the Virtual Block Detector (VBD) depending on the information it receives from the environment:

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

The state of a VSS can be one of the four states:

- *Free*: there is no train on the section.
- *Unknown*: there might be zero or more trains on the section.
- *Ambiguous*: there might be one or more trains on the sections.
- *Occupied*: there is one train on the section.

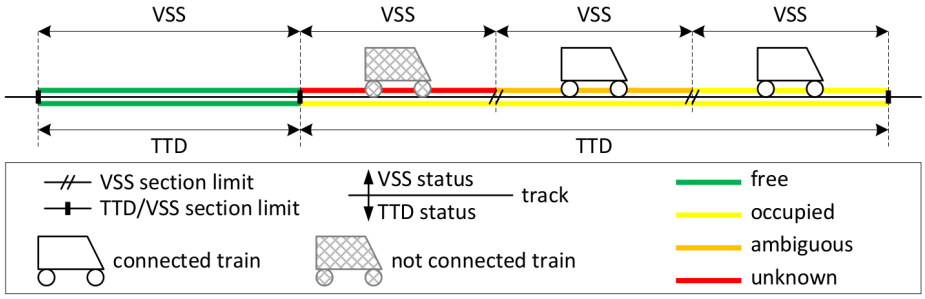

Fig. 1. Section conventions (taken from [4])

The RBC uses free sections to calculate the MA, while the other states are necessary for example to mitigate against possible roll-back of disconnected trains, and to optimise the use of sections in a safe manner.

The transitions from one state to the other can only happen under certain conditions, which are represented as guards in the Event-B model. The VSS state machine is fully connected and the transition table in the specification presents 12 transitions, some of which decomposed into different alternatives.

There are also explicit events to model the expiry of started timers. Additionally, each transition in the VSS statemachine is modelled as an event, we also explicitly model the start and the completion of the statemachine run.

For example take transition #4A which has only one condition: "TTD is free". However, in Event-B this event is modelled as follows:

```
event 4A_unknown_free refines 4_unknown_free
  any vss // generated class instance
  where
    @isin_unknown: vss ∈ unknown
    @grd1: Sections∼(vss) ∉ occupiedTTD
    @grd2: startVSSUpdate = TRUE
    @grd3: vss ∉ updatedVSS
  then
    @act1: updatedVSS := updatedVSS ∪{vss}
    @leave_unknown: unknown := unknown \{vss}
    @enter_free: free := free ∪{vss}
    @act_disconnectProp: disconnectPropagationTimer(vss) := Idle
    @act_integProp: integrityLossPropagationTimer(vss) := Idle
  end
```

Such events can be difficult to validate due to the complexity of the conditions which are difficult to explain to domain experts, hence the need to bridge the gap between domain experts and formal modelling experts.

3 Model Validation

Our approach was to create executable acceptance tests on the plain Event-B level first and to switch to the visual and thus easier to comprehend iUML-B level afterwards. This procedure gives us the opportunity to solve some low-level technical challeges (described in more detail later in this chapter) first and then to deal with the gap between the domain model and the formal model.

3.1 Acceptance Tests for Event-B models.

Gherkin. Gherkin [9] is a language that defines lightweight structures for describing the expected behaviour in plain text as a collection of features, readable by both domian experts and developer, yet still automatically executable.

A feature is a description of one single piece of business value, best structured as a story "*As a* ≪role≫*I want* ≪feature≫*so that* ≪business value≫", which gives an answer to three fundamental questions – *who* requires *what* and *why*.

The features contain a list of scenarios, every scenario representing one use case. In the simplest case the scenario also contains the test data and thus represents an individual test case. A scenario outline describes a group of similar useage scenarios and contains placeholder for the particular test data specified as a list of examples, each data set representing one individual test case.

Each scenario consists of steps describing the interaction with the system under tests: "Given ≪preconditions≫When ≪interaction≫Then ≪postconditions≫", providing an initial state for the test, test input (execution trigger) as well as expected output, the observable outcome shall be compared with.

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).

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 shall, however, still be contained in the features.

Event-B. Event-B [1,5] is a formal method for system development, supported by the Rodin Platform (Rodin) [2], an extensible open source toolkit. A machine in Event-B corresponds to a transition system where *variables* represent the state and *events* specify the transitions.

Cucumber for Event-B. In the scope of this project we have developed Cucumber for Event-B as a custom Cucumber extension, which allows to execute Gherkin scenarios on an Event-B model. It is a collection of step definitions providing means for the Event-B state space traversal:

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 parameters constraints.
Then *event* "≪name≫" *with* "≪formula≫" *is enabled*/*disabled*
Check if the given event with the given parameters constraints is enabled/disabled.
Then *formula* "≪formula≫" *is TRUE*/*FALSE*
Check if the given formula evaluates to TRUE or FALSE.

An essential property of acceptance tests is reproducibility. The user shall assure that the tested machine is deterministic and, if not, refine it further.

Cucumber for Event-B can be found under https://github.com/tofische/cucumber-event-b and has been released under Eclipse Public License 2.0.

Environment definition. Let us deal with the very basic scenario of a train entering the controlled section.

```
Background:
  Given machine
  When fire event "VBD_start_vss_update"
  And fire event "4A_unknown_free" with "vss=VSS11"
  And fire event "4A_unknown_free" with "vss=VSS12"
  And fire event "4A_unknown_free" with "vss=VSS21"
  And fire event "4A_unknown_free" with "vss=VSS22"
  And fire event "4A_unknown_free" with "vss=VSS23"
  And fire event "4A_unknown_free" with "vss=VSS31"
  And fire event "4A_unknown_free" with "vss=VSS32"
  And fire event "4A_unknown_free" with "vss=VSS33"
  And fire event "VBD_vss_update_complete"

Scenario: Enter HL3 area
  When fire event "ENV_enter_HL3_area" with "tr=TRAIN1"
  And fire event "VBD_start_vss_update"
  And fire event "1A_free_unknown" with "ttd=TTD10 & vss=VSS11"
  And fire event "1A_free_unknown" with "ttd=TTD10 & vss=VSS12"
  And fire event "self_free" with "vss=VSS21"
  And fire event "self_free" with "vss=VSS22"
  And fire event "self_free" with "vss=VSS23"
  And fire event "self_free" with "vss=VSS31"
  And fire event "self_free" with "vss=VSS32"
  And fire event "self_free" with "vss=VSS33"
  And fire event "VBD_vss_update_complete"
  Then is TRUE formula "free = {VSS21,VSS22,VSS23,VSS31,VSS32,VSS33}"
  Then is TRUE formula "occupied = {}"
  Then is TRUE formula "ambiguous = {}"
  Then is TRUE formula "unknown = {VSS11,VSS12}"
  When fire event "ENV_start_of_mission" with "tr=TRAIN1"
  # ...
```

This example reveals the fact, that there are two kinds of events which must be treated differently. The environment events represent some relevant change in the environment and are thus triggered from outside of the modeled system (in our case through the tests). The system events on the other side represent the reaction of the modeled system to the external stimulus and shall therefore be considered as an implementation detail not prescribed by the acceptance tests.

The acceptance tests being of black box nature shall contain environment events only. Nevertheless, when running the tests, the system must be given the opportunity to fire all internal events according to the assumed execution strategy. For our purposes run to completion semantic is sufficient – after an environment event (according to the test scenario) fires, the particular step definitions shall automatically trigger all enabled internal events until the system stabilizes and only environment events are enabled. However, this requires some kind of naming convention (e.g. event name prefix) which allows the steps to distinguish the event kind (environment of system).

The previous example would be then reduced to following snippet, including domain events only:

```
Scenario: Enter HL3 area
  When fire event "ENV_enter_HL3_area" with "tr=TRAIN1"
  And fire event "ENV_start_of_mission" with "tr=TRAIN1"
  # ...
```

Event selection. During the refinement process an event is often decomposed into different alternatives, e.g. ENV_exit_HL3_area into ENV_exit_HL3_area and ENV_exit_HL3_area_free_ttd. The acceptance tests shall not be aware of this decomposition, so that they may reference an abstract event, ENV_exit_HL3_area in this case. The step definitions shall then descend the refinement hierarchy and select an appropriate enabled concrete event. This process is unique as long as all concrete refinements of one abstract event are disjunct, otherwise the event selection fails and the model must be adjusted. In order to utilize this capability it might be necessary to rename decomposed events so that they can be clearly distingished from the refined abstract ones.

Timeouts. While unsolicited environment events (caused by some unexpected change in the environment) may occur at any time, answers to previously issued system command shall happen within a defined time period, otherwise the system shall assume an error in the environment and process an appropriate corrective action.

There are several techniques how to model time. If both events (answer and timeout) are enabled simultaneously, the environment (in our case the acceptance tests) must be able to choose, which situation happens. This approach simplifies the model, however the acceptance tests must be aware of the timeout names.

```
# ...
When fire event "VBD_ghost_timer_expires" with "ttd=TTD10"
# ...
```

Another possibility is to consider the timeouts as an internal model concept, and only trigger the time progress (ticks) by the environment. However, the explicit notion of time clutters the model and has therefore been omitted from our model and left for further analysis.

Data. While the event parameters are often simple values, the attribute values may have complex types. This raises the issue of how to describe such data for the setup of constants on one side and for the attribute value checks on the other side. We want to represent the data in a table form, but we have to overcome different viewpoints: class instance groups the values of all attributes (row by row), while in Event-B one variable represents the value of one attribute for all instances (column by column). This is still an open point left for future work.

There is no technical difference between attributes and associations. However, there is a logical distinction between an attribute and an association, which shall be respected by the test language.

3.2 Acceptance Tests for iUML-B models.

iUML-B. 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 are used to describe the behaviour.

It is desirable to express the acceptance tests in terms of the domain model so that domain experts who are not familiar with the formal notations can easily create and validate them.

iUML-B [6,7,8], 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, with automatic generation of Event-B formal models. The iUML-B is a formal notation which is much closer to the domain model and makes therefore the formal models more visual and thus easier to comprehend.

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.

A state-machine automatically generates Event-B data elements (sets, constants, axioms, variables, and invariants) to implement the states, and contributes additional parameters representing the state machine instance, as well as guards and actions representing state changes to existing events elaborated by transitions. State-machines support nested states (hierarchical state machiness) and may be also *lifted* to the instances of a class so that the behaviour of each instance of the class is modelled by an independent instance of the state-machine.

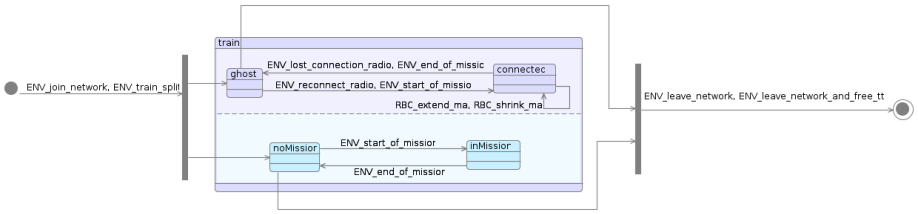

Fig. 2. State machine of train states

Cucumber for iUML-B. Cucumber for iUML-B provides a collection of step definitions translating the iUML-B constructs into the corresponding underlying Event-B model elements (events and variables), allowing the acceptance tests to use the the notation provided by the iUML-B. The acceptance tests can then refer domain elements like classes and their methods, attributes and associations as well as state machine states and transitions.

However, this requires a great deal of discipline on the part of modelers, as only the strict adherence to the Domain Driven Design principles, especially a rigorous compliance to an ubiquitous language shared between the domain experts and the formal modeling experts is crucial, as each deviation leads to failed tests and hence to manual rectifications.

The following steps are defined for validating state-machines:

Given *state machine* "≪name≫:≪inst≫"
: Preset the given instance of the given state machine.

When *trigger transition* "≪trans≫"
: Trigger the given state machine transition.

Then *transition* "≪trans≫" *is enabled/disabled*
: Check if the given state machine transition is enabled/disabled.

Then *is in state* "≪state≫"
: Check if the state machine is in the given state.

The following steps are 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/disabled*
: Check if the given class instance method is enabled/disabled.

Then *attribute* "≪attr≫" *is* "≪value≫"
: Check if the given class instance attribute 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.

4 Conclusion

In this paper we have discussed how the BDD principles can be applied to the formal model validation and also demonstrated the applicability of this approach by providing the Gherkin scenarios for an Event-B/iUML-B formal model of a 'fixed virtual block' approach to train movement control. In summary, we have confirmed the benefits of validating the formal models using the acceptance tests.

In addition we also pointed out, how to close the gap between the informal domain specification and the formal model, thus enabling the domain expert to write acceptance tests in a high-level language matching the formal specification.

Finally, we analysed the advantages of such an approach and proposed measures to mitigate identified drawbacks.

Once validated the acceptance tests can also be used in order to show the conformity of the implementation with respect to the formal model. This transition has also been left for the future work.

Recommendations. We recommend to adopt the BDD methodology already during requirement elicitation phase before modeling activities, as the subsequent adaptation of tests to the existing model is tedious and costly.

Furthermode, we intend to enhance the Cucumber for Event-B framework according to the aforementioned proposals and also integrate it tightly with the iUML-B plugin.

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 https://iktderzukunft.at/en/

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. Dghaym, D., Poppleton, M., Snook, C.: Diagram-led formal modelling using iUMLB for Hybrid ERTMS Level 3. In: Abstract State Machines, Alloy, B, TLA, VDM, and Z: ABZ 2018. vol. 10817, pp. 338–352. Springer (May 2018), https://eprints.soton.ac.uk/417755/
4. EEIG ERTMS Users Group: Principles: Hybrid ERTMS/ETCS Level 3. Ref. 16E042 Version 1A. (July 2017), http://www.ertms.be/sites/default/files/2018-03/16E0421A_HL3.pdf
5. Hoang, T.S.: An introduction to the Event-B modelling method. In: Industrial Deployment of System Engineering Methods, pp. 211–236. Springer-Verlag (2013)
6. 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
7. Snook, C.: iUML-B statemachines. In: Proceedings of the Rodin Workshop 2014. pp. 29–30. Toulouse, France (2014), http://eprints.soton.ac.uk/365301/
8. 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
9. Wynne, M., Hellesøy, A.: The Cucumber Book: Behaviour-Driven Development for Testers and Developers. Pragmatic Programmers, LLC (2012)