Project ADVANCE
Grant Agreement 287563
"Advanced Design and Verification Environment for Cyber-physical System Engineering"

ADVANCE Deliverable D4.2 (Issue 2)

Methods and tools for simulation and testing I

Public Document

March 27, 2013

http://www.advance-ict.eu

Contributors:

Jens Bendisposto, Joy Clark, John Colley, Andy Edmunds,
Lukas Ladenberger, Michael Leuschel, Vitaly Savicks, Harald Wiegard

Reviewers:

Luis-Fernando Mejia, Michael Butler

Contents

This deliverable describes the prototype multi-simulation framework, as well as the result of first experiments. The focus is on describing the simulation of high-level models using the ProB animator and measuring how well the implemented enhancements to this technology meets the performance requirements for high-level simulation. A roadmap for implementing the full-blown multi-simulation framework is also presented, detailing further enhancements to ProB and the requirements for Event-B model code generation to enable high-performance simulation of Event-B models at a more detailed level. [month 12]

Chapter 1

Introduction

1.1 Introduction

As described in the ADVANCE Deliverable 4.1, simulation is the dominant technology for the industrial verification of digital hardware and embedded systems. The use of the Verilog and VHDL modelling languages is widespread in simulation-based digital hardware verification and SystemVerilog and SystemC are used increasingly for the design and simulation of systems. The design and verification of cyber-physical systems, however, introduce new challenges which cannot be easily addressed by existing simulation frameworks. First, it is necessary in a cyber-physical system to model and simulate in the *continuous* as well as the digital domain. Second, the complexity of highly-concurrent systems cannot be addressed by simulation techniques alone. Refinement-based formal methods, such as Event-B help considerably to manage this complexity and to verify that the implemented system meets its specification.

It is not feasible to contemplate developing a single simulation language and verification environment that can meet all the requirements for cyber-physical development and verification. Legacy designs must be re-used and the specialised expertise of developers with existing tool chains leveraged. The primary objective of the ADVANCE multi-simulation framework is to address the needs for different design and verification tools, both discrete and continuous, test-based and formal to co-operate within a single development and verification framework.

1.1.1 Approaches to Cyber-physical System Simulation

No one approach to simulation will meet the needs for cyber-physical development and verification. In the digital hardware verification domain, *cycle-*

based simulation is the dominant technology. However, where delays and deadlines need to be modelled accurately, *discrete time* simulation will be needed. For systems where an accurate, continuous model of the environment is required, *continuous time* simulation must be used. For many cyber-physical systems, a *hybrid* approach to simulation, which employs all of the cycle-based, discrete and continuous time simulation techniques, is required.

Cycle-based Simulation

Synchronous digital hardware can be synthesised directly from a *Register Transfer Level (RTL)* model description where it is assumed that all combinational logic will settle between clock edges and therefore the fundamental unit of time is the clock *tick*. In a cyber-physical development where a high-level description of a system is refined so that hardware is separated from software, the hardware description will then be further refined to a cycle-based description which can be translated directly into an RTL hardware description language model for synthesis. In ADVANCE, it will be possible to simulate and model check the cycle-based description before translation.

Discrete Time Simulation

For hardware/software systems which combine synchronous and asynchronous behaviour, or have safety requirements concerning delays and deadlines, it is necessary to model time more accurately and introduce the notion of actual time and time units. Different components of a system may be modelled with time units of different granularity; the system must therefore be simulated at the time unit of finest granularity.

Continuous Time Simulation

Where no discrete approximation of a system component's behaviour can be realistically modelled, it is necessary to simulate the component in the continuous domain. *Simulink* and *Modelica* provide well-established tools for continuous simulation, together with significant libraries of models. ADVANCE plans, through co-simulation, to leverage these simulators and libraries.

Hybrid Simulation

To provide a complete approach to cyber-physical system simulation, a hybrid approach will be required where cycle-based components interact with discrete and continuous time components within a single simulation environment. Modelica provides a hybrid simulation capability within its language and toolset, but does not support the notion of formal refinement. We therefore intend to combine the advantages of Event-B refinement with the continuous modelling capabilities of Simulink/Modelica in the ADVANCE multi-sim environment.

1.1.2 Multi-sim: the first phase of development

In this first stage of development we focus on cycle-based and discrete time modelling and simulation. We concentrate on the modelling and refinement method for the development and verification of hardware/software systems in an environment that can be modelled using discrete time and the tools that are needed to support this method.

1.1.3 Formalising Simulation

A goal of ADVANCE is to use a formal approach to developing the simulation framework. In a commercial simulator, the semantics of a given simulation is determined by the descriptions of each component model in the design hierarchy to be simulated, as defined in the simulation language reference manual and also the implementation of the synchronisation and communication mechanisms, used by those component models, in the simulator kernel. In ADVANCE, we will develop and refine simulation models which define, formally, not only the components but also the synchronisation and communication mechanisms themselves. The model description itself will therefore define, completely the simulation semantics of that model in the formal language, Event-B. This will enable the ADVANCE user to start with an untimed, high-level model of the system and, using systematic Event-B formal refinement, introduce incrementally more and more timing and synchronisation detail and prove formally that, at each step, the more detailed model is a correct refinement of its abstract parent. At each step it will also be possible to simulate and model check the model description.

1.1.4 Structure of the deliverable

The remainder deliverable is structured as follows

- Chapter 2 describes and formalises various important aspects of simulation and multi-simulation, as well as various ways of modelling time.

- In Chapter 3 we present first preliminary experiments in simulating hybrid systems using Rodin and Event-B.
- The Sim-B approach to multi-simulation of cyber-physical systems is described in Chapter 4. This chapter presents first prototypes on how to effectively simulate cyber-physical systems composed of multiple components.
- Various requirements for simulation quickly arose during the initial phases of the case studies of the project. This led us to realise that we had to re-design part of ProB, and provide a scripting interface. This scripting architecture will also provide the foundation for developing FMI multi-simulation masters and Model testing. This architecture is described in Chapter 5, where a new Groovy-based scripting language for ProB is developed.
- The studies and experiments have led to the roadmap for a multi-simulation framework described in Chapter 6. Led by industrial needs, we decided to build on the industry standard FMI (Functional Mockup Interface).
- The application of the scripting language to Model testing is described in Chapter 7.
- Chapter 8 presents how one can now generate an FMU (Functional Mockup Unit) using code generation for simulation.
- In Chapter 9 we then present first experiments with Modelica and an outlook on the rest of the project.
- Various other ProB tool developments were made in the project, driven by the requirements of the industrial case studies. The final Chapter 10 describes those extensions of ProB and the associated BMotion-Studio, in the context of the case studies and the requirements of, e.g., Chapter 4.

Chapter 2

Formalising Simulation

2.1 Motivation

As described in the ADVANCE Deliverable 4.1, simulation is the dominant technology for the industrial verification of digital hardware and embedded systems. The use of the Verilog and VHDL modelling languages is widespread in simulation-based digital hardware verification and SystemVerilog and SystemC are used increasingly for the design and simulation of systems. The design and verification of cyber-physical systems, however, introduce new challenges which cannot be easily addressed by existing simulation frameworks. First, it is necessary in a cyber-physical system to model and simulate in the *continuous* as well as the digital domain. Second, the complexity of highly-concurrent systems cannot be addressed by simulation techniques alone. Refinement-based formal methods, such as Event-B help considerably to manage this complexity and to verify that the implemented system meets its specification.

It is not feasible to contemplate developing a single simulation language and verification environment that can meet all the requirements for cyber-physical development and verification. Legacy designs must be re-used and the specialised expertise of developers with existing tool chains leveraged. The primary objective of the ADVANCE multi-simulation framework is to address the needs for different design and verification tools, both discrete and continuous, test-based and formal to co-operate within a single development and verification framework.

A further goal of ADVANCE is to use a formal approach to developing the simulation framework. In a commercial simulator, the semantics of a given simulation is determined by the descriptions of each component model in the design hierarchy to be simulated, as defined in the simulation language reference manual and also the implementation of the synchronisation and communication mechanisms, used by those component models, in the simulator kernel. In ADVANCE, we will develop and refine simulation models which define, formally, not only the components but also the synchronisation and communication mechanisms themselves. The model description itself will therefore define, completely the simulation semantics of that model in the formal language, Event-B. This will enable the ADVANCE user to start with an untimed, high-level model of the system and, using systematic Event-B formal refinement, introduce incrementally more and more timing and synchronisation detail and prove formally that, at each step, the more detailed model is a correct refinement of its abstract parent. At each step it will also be possible to simulate and model check the model description.

2.2 Formalising Simulation with Event-B

In ADVANCE, we will develop and refine simulation models which define, formally, not only the components but also the synchronisation and communication mechanisms themselves. The model description itself will therefore define, completely the simulation semantics of that model in the formal language, Event-B.

2.2.1 Temporal Modeling in Cyber-physical systems

The following areas will need to be properly addressed by the temporal modelling capabilities provided by ADVANCE.

Distributed Function and Control

In traditional embedded systems where a single controller interacts with its environment, it is possible to model temporal interaction with the introduction of two modes, *controller mode* and *plant mode*. The system model first evaluates in plant mode, switches to controller mode to deal with any inputs from the plant and then switches back to plant mode so that the plant can respond to the controller outputs. For modern cyber-physical systems with distributed function and control, as shown in Figure 2.1, it is necessary to model the Communication and Synchronisation of the Concurrent Processes in a more general way.

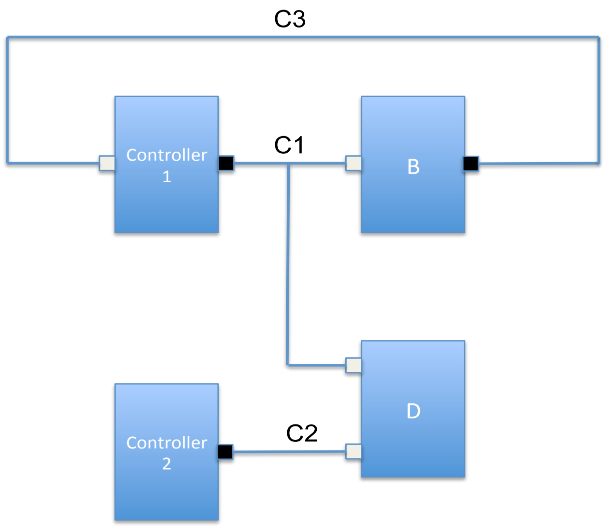


Figure 2.1: Distributed Function and Control

Managing Safety Hazards

In cyber-physical system modelling it is necessary to deal with the safety requirements as well as the functional requirements.

In a simple embedded system, this can be done by introducing two extra modes to the plant/controller mode cycle. In the *detection* mode the controller can model, non-deterministically, the presence of a safety hazard in the system and manage the hazard appropriately. In the *prediction* mode, the controller can model what it thinks the plant's response to a given controller input should be. The controller can then compare the predicted response to the actual response received in *plant* mode.

Again, this modal approach will not work with more complex cyber-physical systems such as that shown in Figure 2.1.

Verifying the relationships between Inputs and Outputs

In the aerospace standard, *DO 178C*, the formal methods supplement states that, for a system verified using formal methods, the certifying authority must be convinced that

- The Outputs fully satisfy the Inputs
- Each Output data item is necessary to satisfy some Input data item (No unintended behaviour)

- The Input/Output specification is preserved by chosen implementation architecture

The DO 178C standard pioneers the introduction of formal methods to complement traditional test-based verification techniques for system certification, describing a method that is applicable to cyber-physical systems in general.

2.2.2 Simulating Formal Models

In ADVANCE the user may start with an untimed, high-level model of the system and, using systematic Event-B formal refinement, introduce incrementally more and more timing and synchronisation detail. It will then be possible to prove formally that, at each step, the more detailed model is a correct refinement of its abstract parent.

Whereas the abstract model(s) may be untimed, the refined models represent concurrent, communicating processes. It will therefore be necessary to introduce the notion of a *tick* in the formal model.

Cycle-based execution

In cycle-based simulation, there is no notion of modelling the actual time in units. There is just the notion of time advancing periodically. In Event-B terms we can therefore differentiate between a sequence of events that occur *before* the *tick* and a sequence of events that occur *after* the *tick*. This has implications for the ADVANCE toolset. For instance, ProB has the notion of the *next state, X,* in it's LTL implementation, which corresponds to the next event evaluation. We need the notion of the *next tick*.

Timed execution of Delays and Deadlines

In the cases where it is necessary to model *delays* and *deadlines* to meet the system requirements, time will need to be modelled properly in integer time units.

2.2.3 Modeling Timing Cycles with Event-B

Component Modes

As described above, for simple models with a single controller, it is sufficient in Event-B to introduce a *Plant* mode and a *Controller* mode. The Plant evaluates, the Controller then evaluates and the next *tick* begins when the

Plant evaluates again. For more complex systems with multiple controllers and distributed plant, the number of modes proliferate and it is necessary to define an ordering on these modes to ensure correct evaluation of the model events. If there is any feedback in the model, as introduced by the connection *C3* in Figure 2.1 above, it will not be possible to define an ordering. A further problem with using component modes is that typically components evaluate at different rates, but all components must be evaluated at the fastest rate whether they need to or not.

Generalised Update/Evaluation Modes

To overcome the need for defining an ordering on Component modes, commercial simulators use a *two list* algorithm which has two, generalised modes: *Update* and *Evaluate*. The algorithm, which is described in detail in deliverable *D4.1*, supports arbitrary topology complexity. No zero-delay communication is allowed between components which prevents *race* and means that components can be evaluated in any order. Components *suspend* between wake-ups and will be evaluated again when a *value change* is received on an input port or a *self wake* matures. Components can therefore evaluate at different rates. The two-list algorithm supports Discrete Time modelling of delays and deadlines, Cycle-based modelling or a combination of both. The algorithm is also therefore suitable for modelling Safety Hazards.

2.2.4 A Simple Arithmetic Example

The Abstract Specification

We first write the abstract specification in Event-B, defining the *Output* as a function of its *Inputs* as shown in Figure 2.2.

```
event AddInc
   any p
   where
     @grd1 p ∈ Inputs
     @grd2 In1(p) ∈ ℕ
     @grd3 In2(p) ∈ ℕ
   then
     @act1 v ≔ In1(p) + In2(p) + 1
  end
```

Figure 2.2: The Abstract Specification

At this level, the model is untimed. The whole arithmetic operation is performed atomically. The single event *AddInc* takes two natural numbers as inputs, adds them together, adds one to the result and stores this result in the variable v The *Inputs* are defined in the Event-B *context* as shown in Figure 2.3. *Inputs* is a subset of the set *Parameters*. *In1* and *In2* define mappings from *Inputs* to the Natural numbers.

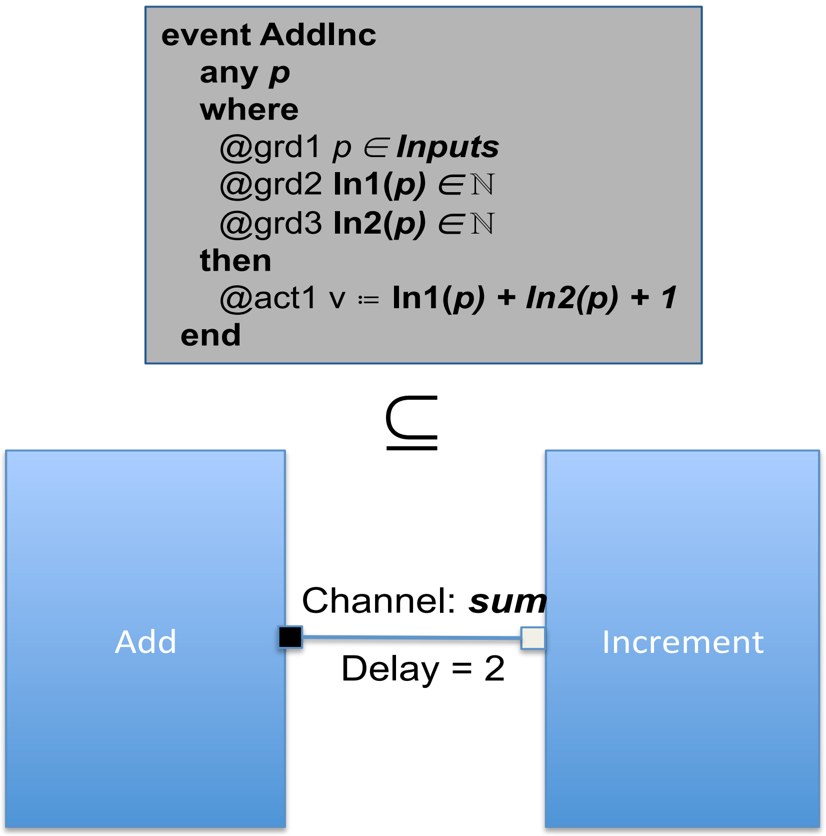
```
constants Inputs In1 In2

sets Parameters

axioms
  @axm1 Inputs ⊆ Parameters
  @axm2 In1 ∈ Inputs → ℕ
  @axm3 In2 ∈ Inputs → ℕ
end
```

Figure 2.3: The Input Specification

The First Refinement

In this refinement, we introduce the implementation architecture. We wish to implement the arithmetic operation as a *pipeline*. The first stage of the pipeline is represented by a component which adds the two input values together and writes the result to the channel. The second component reads the value from the channel when it arrives, adds one to it and stores the value in v. We model the abstract channel as having a delay of two time units, as shown in Figure 2.4.

Does the chosen Architecture Implement the Abstract Specification? Can we verify that the relationship between the Output and the Inputs, is preserved, as required by DO-178C?



```
event AddInc
   any p
   where
     @grd1 p ∈ Inputs
     @grd2 In1(p) ∈ ℕ
     @grd3 In2(p) ∈ ℕ
   then
     @act1 v ≔ In1(p) + In2(p) + 1
  end
```

Figure 2.4: The Pipelined Architecture

Modelling Channels with Delay in Event-B

We model a Channel as a *Set of Schedules*. A *Schedule* comprises a *Delay*, a *Value* (optional) and the *Input Values* that correspond to the *Output Value* (optional)as shown in Figure 2.5. The Input Values are used to prove that the input/output relationship is preserved by the implementation. A channel with schedules that have Delay only no Values can be used for synchronisation purposes.

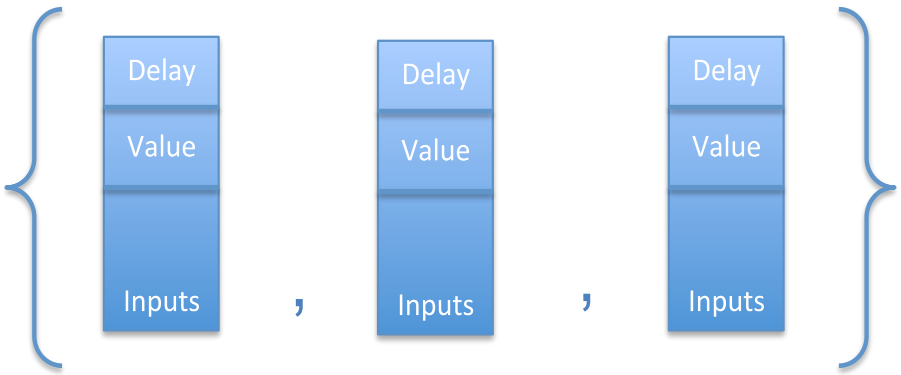


Figure 2.5: The Channel

Writing to a Channel

A Channel *write* is accomplished by creating a new schedule with a delay of at least *one* as shown in Figure 2.6. Multiple Schedules may be added to a Channel. If more than one schedule is present for a given time, one is chosen non-deterministically. It is the subject of future work to explore other options to match the semantics of different commercial simulators. For instance, it would be possible to prevent a component writing multiple schedules to a channel for same time.

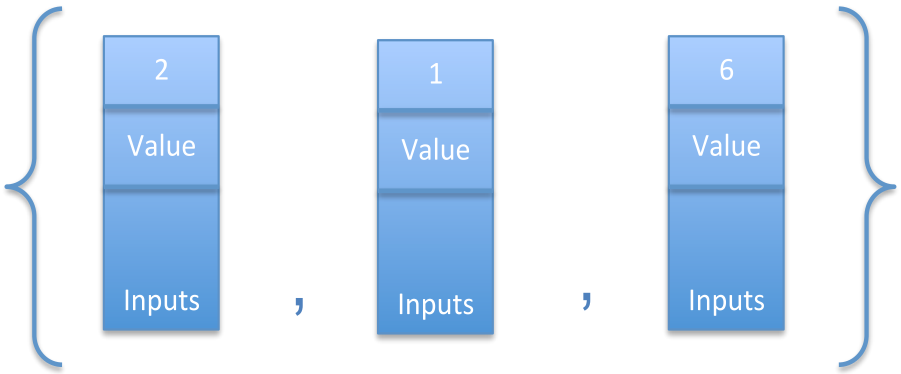


Figure 2.6: Writing to a Channel

The Update/Evaluate Cycle

Update is modelled using a single Event-B event. *Evaluate* is represented by one or more enabled Component Events.

In *Evaluation Mode*, all Components, where one or more of their Input Channels has a schedule with delay 0, as shown in Figure 2.7, resume. One or more Component events are enabled and the Update event is disabled. A Component may change local state and create new Schedules on Output Channels. It must then *suspend*.

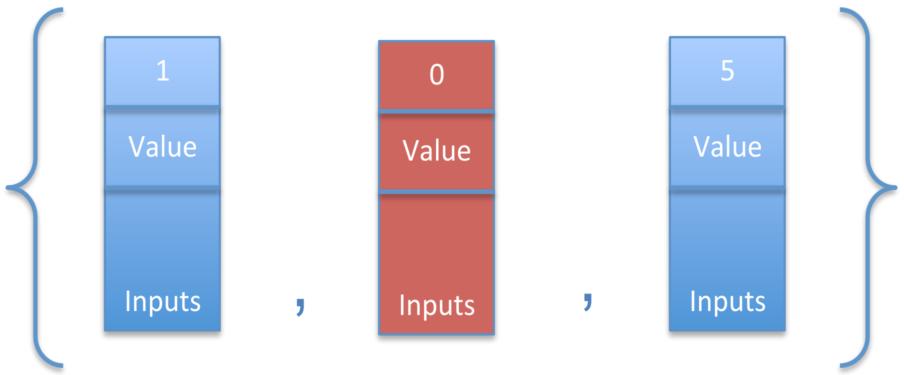


Figure 2.7: Evaluation Mode

When all of the enabled Components have been evaluated, all the Component evaluation events will be *disabled* and the *Update* event *enabled*. Schedules with 0 delay are deleted and all other schedule delays are decremented as shown in Figure 2.8. The current tick is therefore complete The Update Event is re-enabled if no schedule has 0 delay, resulting in another tick.

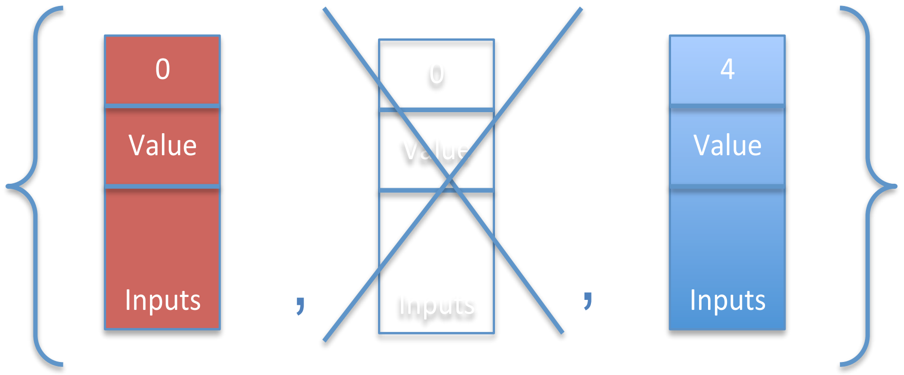
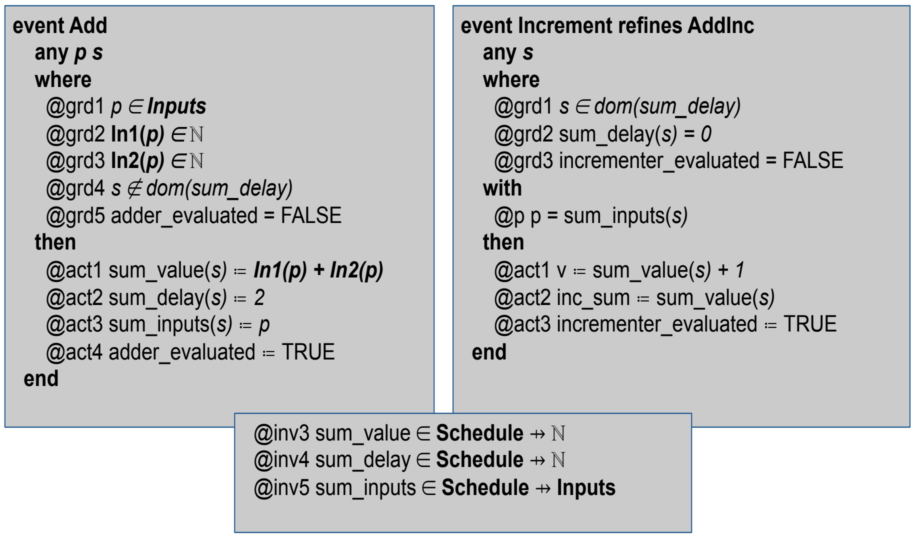


Figure 2.8: Update Mode

Modelling the First Refinement in Event-B

We model the *Adder* and *Incrementer* Components as shown in Figure 2.9.



```
event Add
   any p s
   where
     @grd1 p ∈ Inputs
     @grd2 In1(p) ∈ ℕ
     @grd3 In2(p) ∈ ℕ
     @grd4 s ∉ dom(sum_delay)
     @grd5 adder_evaluated = FALSE
   then
     @act1 sum_value(s) ≔ In1(p) + In2(p)
     @act2 sum_delay(s) ≔ 2
     @act3 sum_inputs(s) ≔ p
     @act4 adder_evaluated ≔ TRUE
  end
```

```
event Increment refines AddInc
   any s
   where
     @grd1 s ∈ dom(sum_delay)
     @grd2 sum_delay(s) = 0
     @grd3 incrementer_evaluated = FALSE
   with
     @p p = sum_inputs(s)
   then
     @act1 v ≔ sum_value(s) + 1
     @act2 inc_sum ≔ sum_value(s)
     @act3 incrementer_evaluated ≔ TRUE
  end
```

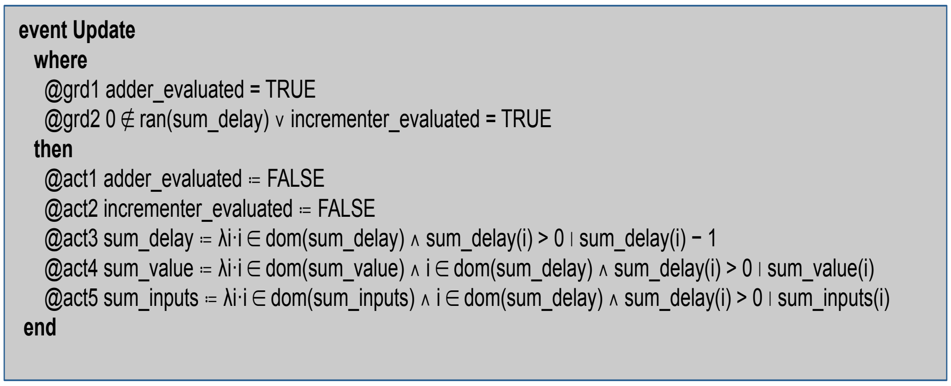
```
@inv3 sum_value ∈ Schedule ⇸ ℕ
@inv4 sum_delay ∈ Schedule ⇸ ℕ
@inv5 sum_inputs ∈ Schedule ⇸ Inputs
```

Figure 2.9: The Add and Increment Components

The channel is modelled as three functions, *sum_ value*, *sum_ delay* and *sum_ inputs* which represent the *records* in the schedule.

The event *Add* is a new event which refines *Skip*. It takes the Inputs and a fresh schedule as parameters, adds the input values together and writes the result to the *value* record of the schedule, writes a delay of 2 to the *delay* record, and writes the Input parameter to the *inputs* record. The *adder_ evaluated* flag is set to *FALSE* to indicate that the adders has completed its evaluation. The *Increment* event refines the abstract event*AddInc*. It is enabled when there is a schedule on the channel which has a delay of 0. It takes the value from the schedule using the *sum_ value* function, adds 1 to it and writes the result to the variable v. It then sets the *incrementer_ evaluated* flag to *FALSE*.

When both Components have evaluated, the *Update* event, shown in Figure 2.10, is enabled.



```
event Update
  where
   @grd1 adder_evaluated = TRUE
   @grd2 0 ∉ ran(sum_delay) ∨ incrementer_evaluated = TRUE
  then
   @act1 adder_evaluated ≔ FALSE
   @act2 incrementer_evaluated ≔ FALSE
   @act3 sum_delay ≔ λi·i ∈ dom(sum_delay) ∧ sum_delay(i) > 0 ∣ sum_delay(i) − 1
   @act4 sum_value ≔ λi·i ∈ dom(sum_value) ∧ i ∈ dom(sum_delay) ∧ sum_delay(i) > 0 ∣ sum_value(i)
   @act5 sum_inputs ≔ λi·i ∈ dom(sum_inputs) ∧ i ∈ dom(sum_delay) ∧ sum_delay(i) > 0 ∣ sum_inputs(i)
 end
```

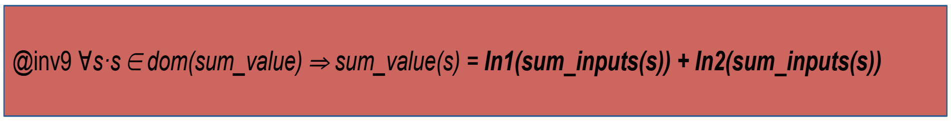
Figure 2.10: The Update Event

The component evaluation flags are set to *FALSE* and all schedules in the channel which have a delay greater than 0 have their delays decremented using a *lambda* function. All schedules with a delay of 0 are removed from the channel.

The *Update* event is now disabled and the event *Add* enabled. The *Increment* is also enabled if there is a schedule with delay 0 on the channel.

Proving that the Timed Implementation is a correct Refinement of the Abstract Specification

To prove that this is, indeed, a correct refinement it is necessary to introduce a *Gluing Invariant*. Because of the way the channels have been modelled as a set of schedules, the Gluing Invariant is readily described as the relationship between inputs and outputs in each schedule, as shown in Figure 2.11.



```
@inv9 ∀s·s ∈ dom(sum_value) ⇒ sum_value(s) = In1(sum_inputs(s)) + In2(sum_inputs(s))
```

Figure 2.11: The Gluing Invariant

With this invariant, all proof obligations that have been generated are discharged automatically by the Rodin tool. The invariant also performs the role of demonstrating that the relationship between inputs and outputs, as required for DO 178C certification, is preserved by the implementation.

The Second Refinement: Removing reference to Input Values

Once the Gluing Invariant has been proved, a further refinement is introduced in which all reference to the input values are removed as they will have no place in the final implementation.

An alternative, Cycle-Based Implementation

In a cycle-based implementation, there can only two schedules on a channel at any time. The schedule representation can be simplified; the *delay* is represented by two boolean variables, *msg_ sent_ on_ sum* and *msg_ rcvd_ on_ sum*, the *value* by integer variables *sum* and *sum_ prime* and the *inputs* by the variables *inputs* and *inputs_ prime*. Two gluing invariants are needed as shown in Figure 2.12.

```
@inv9 msg_rcvd_on_sum = TRUE ⇒ sum = In1(sum_inputs) + In2(sum_inputs)
@inv10 msg_sent_on_sum = TRUE ⇒ sum_prime = In1(sum_inputs_prime) + In2(sum_inputs_prime)
```

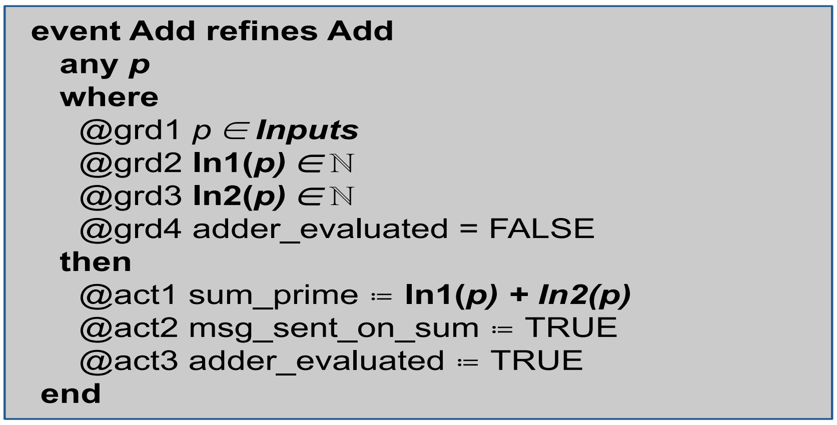
Figure 2.12: The Cycle-Based Gluing Invariant

The second refinement, where the input variables have been removed, is shown below. Figure 2.13 shows the cycle-based *Update* event,

```
event Update refines Update
  where
   @grd1 adder_evaluated = TRUE
   @grd2 msg_rcvd_on_sum = FALSE ∨ incrementer_evaluated = TRUE
  then
   @act1 msg_rcvd_on_sum ≔ msg_sent_on_sum
   @act2 sum ≔ sum_prime
   @act3 msg_sent_on_sum ≔ FALSE
   @act4 adder_evaluated ≔ FALSE
   @act5 incrementer_evaluated ≔ FALSE
 end
```

Figure 2.13: The Cycle-Based Update Event

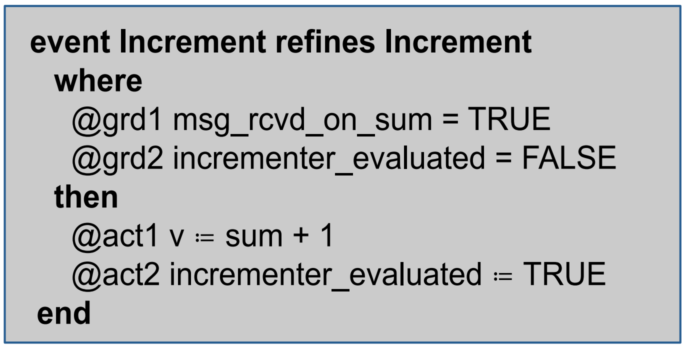
Figure 2.14 shows the cycle-based *Add* event,



```
event Add refines Add
  any p
  where
    @grd1 p ∈ Inputs
    @grd2 In1(p) ∈ ℕ
    @grd3 In2(p) ∈ ℕ
    @grd4 adder_evaluated = FALSE
  then
    @act1 sum_prime ≔ In1(p) + In2(p)
    @act2 msg_sent_on_sum ≔ TRUE
    @act3 adder_evaluated ≔ TRUE
 end
```

Figure 2.14: The Cycle-Based Add Event

and Figure 2.15 shows the cycle-based *Increment* event,



```
event Increment refines Increment
  where
    @grd1 msg_rcvd_on_sum = TRUE
    @grd2 incrementer_evaluated = FALSE
  then
    @act1 v ≔ sum + 1
    @act2 incrementer_evaluated ≔ TRUE
 end
```

Figure 2.15: The Cycle-Based Increment Event

2.2.5 Formalising Simulation: Summary

- Update/Evaluate Simulation semantics can be formalised in Event-B
- Event-B component models can be simulated/co-simulated with third party simulators
- Event-B refinement supports naturally the DO-178C requirement to verify the relationship between Inputs and Outputs at Specification and Implementation Level
- Update/Evaluate modes provides a suitable basis for a Formal Safety Analysis

Chapter 3

First Experiments on Simulating Hybrid Systems using ProB

We have experimented with several case studies, notably those from [ASZ12]. We present two of them below.

3.1 Train Controller

It was relatively straightforward to take the train controller from [ASZ12] and animate it using ProB, even without real number support. The model itself came from the book [Pla10].

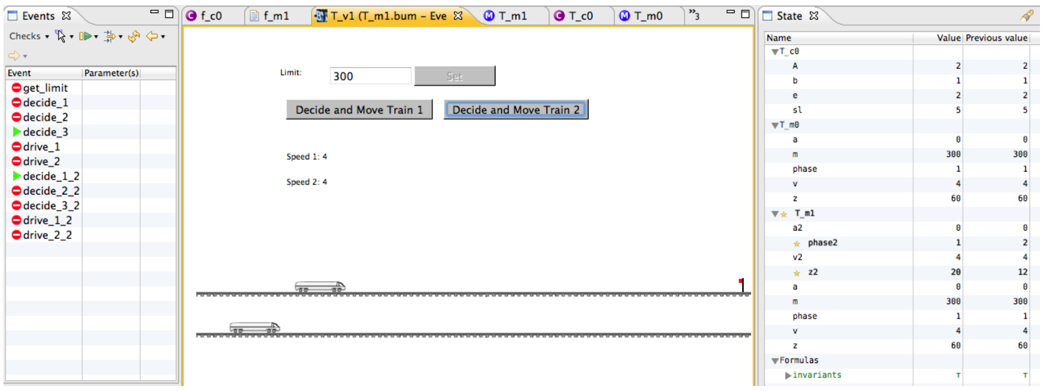
Basically, axiom 5 ($\forall x, y. \neg y = 0 \Rightarrow y * (x/y) = x$) was marked as a theorem rather than an axiom.

Animation with ProB has revealed the following issues with the model:

- Visualization shows that train 1 and 2 can get the same position (invariant 6 of the model is $z - z2 >= 0$, but it should be $z - z2 > 0$ or better $z - z2 > train\_length$).
- The constant sl must be set to at least a value of 4 so that the trains can be moved at all.

3.2 Collision Avoidance Protocol

We have managed to animate the first level of the collision avoidance protocol model from [ASZ12] (see Figure 3.2). The second level of refinement requires access to the trigonometric functions. We have already made such functions

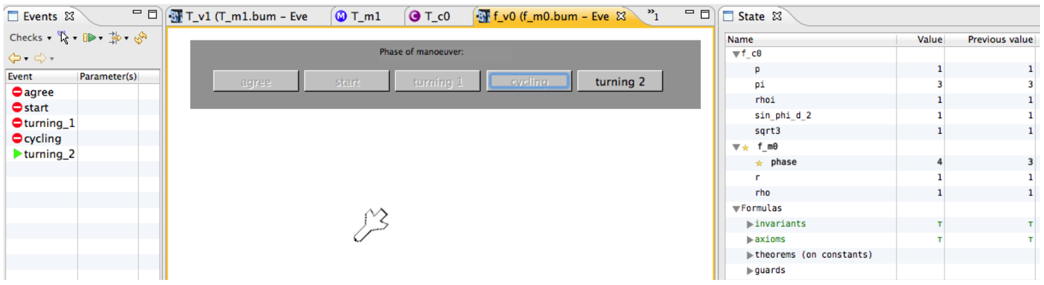


Event	Parameter(s)
get_limit	
decide_1	
decide_2	
decide_3	
drive_1	
drive_2	
decide_1_2	
decide_2_2	
decide_3_2	
drive_1_2	
drive_2_2	

Name	Value	Previous value
T_c0		
A	2	2
b	1	1
e	2	2
sl	5	5
T_m0		
a	0	0
m	300	300
phase	1	1
v	4	4
z	60	60
T_m1		
a2	0	0
phase2	1	2
v2	4	4
z2	20	12
a	0	0
m	300	300
phase	1	1
v	4	4
z	60	60
Formulas		
invariants	T	T

Figure 3.1: ProB and BMotion Studio animation of the hybrid train controller

available within ProB but thus far this only works for classical B models. We are in the process of making these functions available as a mathematical extension (a theory for the theory plug-in).



Event	Parameter(s)
agree	
start	
turning_1	
cycling	
turning_2	

Name	Value	Previous value
f_c0		
p	1	1
pi	3	3
rhoi	1	1
sin_phi_d_2	1	1
sqrt3	1	1
f_m0		
phase	4	3
r	1	1
rho	1	1
Formulas		
invariants	T	T
axioms	T	T
theorems (on constants)		
guards		

Figure 3.2: ProB and BMotion Studio animation of the hybrid collision avoidance protocol model

3.3 Related Work: DESTECS

DESTECS [BLV$^+$10] (Design Support and Tooling for Embedded Control Software) is a tool platform and method for developing embedded control software. It combines continuous-time (CT) models of physical systems with discrete-event (DE) models of controllers through co-simulation and co-modeling. The modeling and simulation of the discrete time-event part takes place in the Overture tool [LBF$^+$10]. Overture is an integrated development environment for modeling and simulating VDM [Jon90] models. The modeling and simulation of the physical part of the system takes place in the

20-sim tool. 20-sim allows to simulate the behavior of dynamic systems, such as electrical, mechanical and hydraulic systems. Since, 20-sim and Overture are in principle two independent tools, the task of the DESTECS platform is to bring both tools together by orchestrating their simulators. Similar to the DESTECS platform, the aim of the ADVANCE project is to combine separate simulation tools in a seamless way. Beside a closely-coupled interface with ProB, the ADVANCE project will also support links with external simulators.

Chapter 4

Sim-B

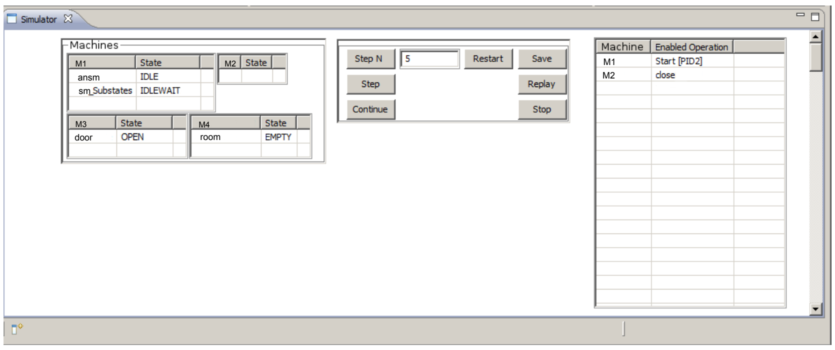
4.1 Overview

The ADVANCE project aims to facilitate the modelling and simulation of hybrid systems. That is, systems that involve software and hardware interaction. Event-B [Abr10] makes use of modelling concepts based on set-theory, predicate logic, and refinements for step-wise development of formal models. System properties are specified in the form of invariant predicates. The Rodin tool [ABHV06] generates proof obligations which have to be discharged in order to show that invariants hold after update actions have been performed. Event-B is a state-based approach, rather than process-based, but we are still able to gain a good understanding of behavioural aspects of the system using the ProB animator [LB03]. The ProB animator can be used to view a model's behaviour in a step-by-step fashion, by manually selecting which event should update the state. ProB can also be used as a model-checker, to perform state-space exploration, looking for deadlocks and invariant violations, and to perform LTL model-checking. To achieve this, the animator randomly selects from a list of enabled states, to updates the state.

4.2 Simulating Cyber-Physical Systems

In the ADVANCE project, we plan to introduce the ability to animate a number of machines with ProB. We also aim to facilitate interaction with external software; be it in the form of 3rd party simulators, our own generated simulation code, or the generated deployable code [EB11]. As a first step we are performing some experiments, and developing a GUI, as part of a simulation plug-in that we call *Sim-B*. The Sim-B GUI will provide an interface for driving simulations, and for making visible a meaningful subset of the available information. We are aware of the potentially large amount of information that will be available for presentation to the user, obtaining (by filtering) a meaningful subset will assist in being able to understand and manage the simulations. It is envisaged that we will be able to provide filters for some of this information, such as providing tables to display a subset of each machine's state, similar to Fig. 4.1. (First developments have already been achieved; see Section 10.1.) Here we see four machines, M1 to M4, where information is displayed about their state-machines' current states. However, the choice of what to display at any time is likely best left to the user, so a configuration menu with choice of whether to display a machines state-machines' current states, and/or to select a *watch* on particular variable, would be most useful. We would also wish to apply breakpoints, to events, so that a simulation will run until a breakpoint event has completed. This feature should be reasonably easy to add to Sim-B. But an interesting issue, that requires further exploration, is what happens to tasks/processes, running in 3rd party simulators and other executables, when a breakpoint is hit. This will be explored in future work, as will be the issue of interfacing with external programs, in general.

In our approach, we can model the environment, and the deployable system in a modular fashion. We can begin with a single all-encompassing model, which is later decomposed [But09] using the decomposition plug-in [SPHB10]; or, it can be done by combining machines, using the composed machine plug-in [Sil11]. Either way, it results in the use of a composed machine component. A composed machine can be viewed as a container for a number of machines; it describes how the machines interact, by recording



M1	State
ansm	IDLE
sm_Substates	IDLEWAIT

M2	State

M3	State
door	OPEN

M4	State
room	EMPTY

Machine	Enabled Operation
M1	Start [PID2]
M2	close

Figure 4.1: A Mock-up GUI for SimB

the synchronizations of events. See the following:

> **COMPOSED MACHINE** $Sys0$
> **INCLUDES** $b1,\ Prod1,\ Cons1$
> **COMPOSES EVENTS**
> $INITIALISATION \triangleq$
> **COMBINES EVENT** $b1.INITIALISATION \parallel$
> $Prod1.INITIALISATION \parallel\ Cons1.INITIALISATION$
> $Put \triangleq$
> **COMBINES EVENT** $Prod1.Put \parallel\ b1.Put$
> $Get \triangleq$
> **COMBINES EVENT** $b1.Get \parallel\ Cons1.Get$
> $Produce \triangleq$
> **COMBINES EVENT** $Prod1.Produce$
> $Consume \triangleq$
> **COMBINES EVENT** $Cons1.Consume$
> **END**

We see here that the composed machine $Sys0$ contains three machines *b1*, *Prod1*, and *Cons1*. Each machine models a buffer, producer, and consumer, respectively. We can see that the producer *Prod1* and buffer *b1* synchronize by combining the events *Prod1.Put* and *b1.Put*. The net effect is realized by conjoining guards of the individual events, and parallel composition of the actions, to insert a value into a non-full buffer. Similarly the *Get* event models the read and removal of the value from a non-empty buffer. Although the machines remain as individuals in the project workspace, a single, statically-checked model is produced, which contains all of the state and events composed as described above. When working downstream of the composed machine, we therefore have the choice whether to make use of the individual models or the single statically checked model.

4.3 Handling Heterogeneity

During preliminary investigations the Düsseldorf team have made some decisions about how the ProB tool will be engineered. As described in Chapter 5, the tool will make use of the Groovy language [KGK$^+$07]. However, we are exploring alternative approaches to driving the simulation. For instance, when

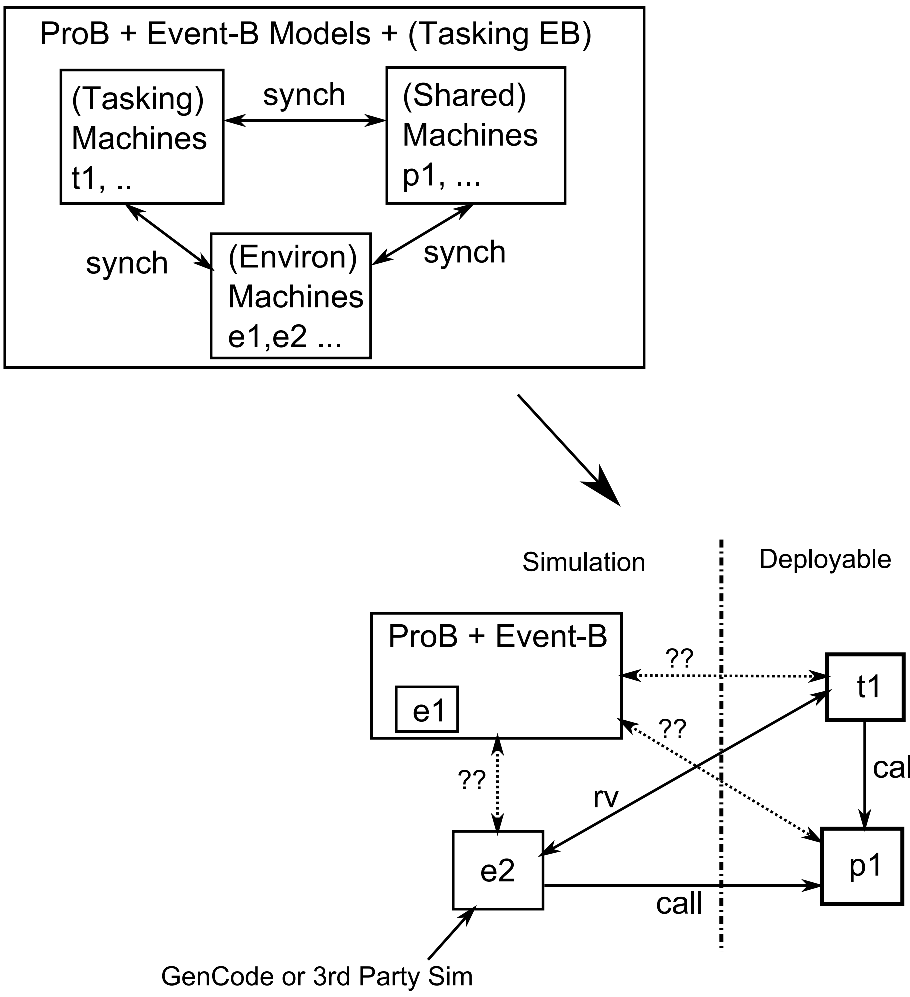

ProB + Event-B Models + (Tasking EB)
(Tasking) Machines t1, ..
synch
(Shared) Machines p1, ...
synch
(Environ) Machines e1,e2 ...
synch
Simulation
Deployable
ProB + Event-B
e1
??
??
??
t1
call
rv
e2
p1
call
GenCode or 3rd Party Sim

Figure 4.2: SimB Simulation - Artefacts

generating all of the code from the formal model, we can perform simulation using the generated environment simulation-code and the actual deployable-code [ACB12], without the need for ProB at all. The pro's and cons of the approach need to be explored, as is the possibility of integrating ProB with generated code (both environment simulation-code and deployable-code), together with 3rd party simulators such as simulink. The artefacts involved in simulation can be seen in Fig.4.2.

The diagram in Fig.4.2 shows an abstract development, at the top-left, containing various components that are modelled, and animated, totally within Event-B and the ProB environment. The bottom right is a representation of some more concrete artefacts used in simulation. The task $t1$, and the protected object $p1$, on the right side of the vertical line, are deployable components, generated by the code generation plug-in. On the left of the vertical line we have simulation components, the environment simulation module $e2$ could be generated by the code generation plug-in (or in fact $e2$ may be a 3rd party simulator), while $e1$ remains animated by ProB. Note that the synchronized event model of communication, in the abstract model, is replaced by concrete subroutine calls, and rendezvous (rv) style communication, in the simulation. So, at some stage of the abstract development it is likely that one or more of the modelled components will be replaced, with something that more accurately represents the actual system being developed. We will need to understand how these artefacts interface with ProB, and the other parts of the simulation to achieve this. The best way to proceed, from the top-left animation of the abstract development, to the bottom-right simulation, the way of interfacing with each of the simulation components, and the method of driving the simulation itself, is the subject of future work.

An overview of the issues relating to cyber-physical systems can be found in [DLV12]. Of particular relevance, is their approach to modelling of heterogeneous systems; that is, modelling with a variety of discrete-event based models and continuous-time models which they refer to as models of computation. This work has its roots in an approach described as a framework for models of computation [LSV98]. The paper describes a model with processes, signals and connections. The semantics of the approach are denotational, set-based and involve named events (tags). Signals are defined by a set of events $e \in T \times V$ where T is a set of tags and V is a set of values. Signals can be combined into a set of tuples, and the tuples can be used to define processes. Processes may be composed using connections, where some signals become hidden. Discrete-event systems are described, based on a timed tag system. This means event tags are used as time-stamps that are used to define ordering between process, and synchronizations are based on equivalent event names. A later paper [LLEL05] describes how the heterogeneous systems, which make up cyber-physical systems, can be combined, using a formal approach. The realisation of the theory is to be found in the *Vergil* tool, of the Ptolemy project [Thec]. The tool makes use of a graphical modelling interface (among others) for simulating cyber-physical systems. The features of the tool are quite extensive, but deal mainly with simulation; code generation is only an experimental feature. It has a port to Eclipse but does not seem to follow the Eclipse plug-in style, so would be of limited use for quick integration with Rodin; however, the aims of Ptolemy and Advance seem quite similar.

Chapter 5

ProB Scripting Approach

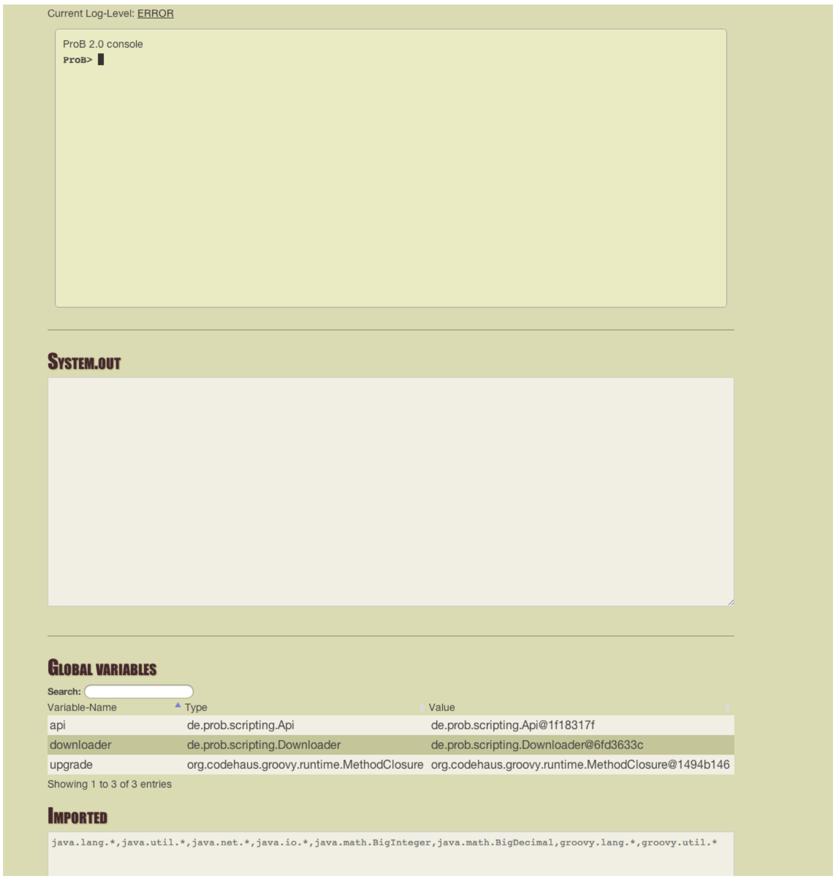
5.1 Current state of the implementation

The current version of the ProB Plug-in for Rodin consists of a component written in Prolog which is wrapped in a Java layer. The Java layer can be extended to enable third-party developers to use ProB within in their own tools. This has been done successfully by several projects. For instance, SAP built a tool on top of ProB [RKW$^+$10] and the UML-B Statemachine Animation plug-in [SS] is also based on ProB. Central to the architecture is a variation of the command pattern [GHJV95]. A command represents a particular calculation that we want to perform in ProB. Examples for commands are “load a model" or “check the invariant for state X". The command implementation has methods to produce a call to the Prolog core and it is also used as a container for the results of the computation. To perform a command, it is sent to a particular object, the animator. The animator is responsible for transmitting the Prolog queries to ProB and extracting the answers to the query.

A third-party developer can combine the provided commands to build up tools or invent new commands by extending both the Java and the Prolog component. However, sometimes this fine grained approach becomes cumbersome. Having a notion of states, events and a state space on the Java side would simplify many tasks.

The new Java API [BCK12] introduces several higher level constructs such as the state space, states, events, formulas (i.e. expressions and predicates that can be evaluated in a given state) and history. The state space is a graph whose vertices are states and edges are events. It is transparently and lazily expanded as new vertices are explored. The state space does not incorporate a notion of a current state. This is captured in the history. A history is a tuple of a path through the state space and a current state that lies on the path. Basically it behaves like a history in a web browser. The new core supports having multiple histories on the same state space. The implementation is, however, still based on the extensible command approach. This means that if the abstractions that are provided are not sufficient it is always possible to access the commands directly.

The new API also provides a development REPL interface (see Fig. 5.1) using the Groovy scripting language. The interface is web based and supports most features that one expects from a modern console such as code completion and a history of commands.


Current Log-Level: ERROR

```
ProB 2.0 console
ProB>
```

SYSTEM.OUT

GLOBAL VARIABLES

Search:

Variable-Name	Type	Value
api	de.prob.scripting.Api	de.prob.scripting.Api@1f18317f
downloader	de.prob.scripting.Downloader	de.prob.scripting.Downloader@6fd3633c
upgrade	org.codehaus.groovy.runtime.MethodClosure	org.codehaus.groovy.runtime.MethodClosure@1494b146

Showing 1 to 3 of 3 entries

IMPORTED

```
java.lang.*,java.util.*,java.net.*,java.io.*,java.math.BigInteger,java.math.BigDecimal,groovy.lang.*,groovy.util.*
```

Figure 5.1: ProB Groovy REPL

To give a feeling of how the console can be used we will show a small script that we can run interactively.

```
// Load an EventB model of the 8-puzzle
// http://en.wikipedia.org/wiki/Fifteen_puzzle
```

```
ProB> model = api.eventb_load('/Puzzle8/MPuzzle8.bum')
([MPuzzle8, CPuzzle8], [SEES=(MPuzzle8,CPuzzle8)])

// Extract the Machine from the model
ProB> mch = model.getComponent('MPuzzle8')
de.prob.model.eventb.EventBComponent@15ab7626

// Find out which Events are defined in the machine
ProB> mch.getChildrenOfType(BEvent.class)
[INITIALISATION, MoveDown, MoveUp, MoveRight, MoveLeft]

// Find out which variables are defined in the machine
ProB> mch.getChildrenOfType(Variable.class)
[board, count]

// Retrieve the model's state space
ProB> s = model.getStatespace()
([root], [])

// Get the root state of the state space
ProB> r = s.root
root

// Call a sequence of two arbitrary Events.
ProB> v1 = r.anyEvent().anyEvent()
1

// Get the value of the model's variable called count.
ProB> v1.value("count")
0

// Find out which events are enabled in state v1
ProB> s.getOutEdges(v1)
[3=[1,3], 2=[1,2]]

// Execute the MoveDown event
v2 = v1.MoveDown()
2

// Get the value of count in a different state
ProB> v2.value("count")
1
```

Using meta programming in Groovy, we can turn events into methods. The methods take an extra argument, a predicate string that can be used to restrict the parameters in case of non-determinism. If we leave out the predicate, ProB will use TRUE. ProB will try to find parameters such that the guard and the additional predicate are true and will execute the event with some arbitrary solution it finds. We can also use the anyEvent method to randomly execute an event. The method enables the restriction of the random choices by providing either a list of event names or a regular expression that is matched against the event names.

The state space is a graph that saves the information about the different states and events that have been calculated. In the example script, we directly used the state space. In the context of animation, we require another abstraction, a history. A history stores the trace of the different events that are executed during an animation process. To begin the animation process it is only necessary to specify the desired state space, and then animation can take place by adding the desired transition from the current state.

A first attempt at synchronizing components (using B-style composition) is also in place. This is accomplished by combining histories over the component's state spaces into a synchronized history. If an event is synchronized, it can only be added to the synchronized history if it is possible to execute the event with the same parameters in all the underlying components. Non-synchronized events can take place arbitrarily. Basically, the synchronized history defines a single valid interweaving of the components but it can also be used in principle to model check the system.

5.2 Ongoing activities

Currently, functionality for the Classical B and CSP formalisms are available in a stand-alone webconsole. This web console is also integrated into an Rodin plug-in so that the Event-B specification is also supported.

It seems reasonable that Groovy is well suited for generating test cases if we can lift the constraint solving capabilities of ProB into the scripting language. This will allow the specification of algorithms for test case generation without deep knowledge of the Prolog engine. This will allow third-party contributors to define their own test case generators.

A detailed description of the requirements for the new ProB version is available from `http://goo.gl/v9j9u`. Note that the requirements document includes many features that are not related to the Advance project. It also does not yet include all requirements imposed by Advance.

Chapter 6

Roadmap for multi-simulation framework based on FMI

6.1 Functional Mock-up Interface

We decided to use the Functional Mock-up Interface (FMI) Standard [Mod] for our implementation of a multi-simulation framework. This will allow us to integrate artefacts called Functional Mockup Units (FMU) into Event-B simulations. These FMUs can be generated by any tool that supports this standard. Many simulation tools[1] such as MATLAB/Simulink or even Microsoft Excel support the FMI standard, as it ensures flexibility and cross-platform execution.

FMUs are coordinated by an FMI master. An individual FMU is a zip archive that contain at least a shared library which implements the Functional Mock-up Interface and an XML document describing the communication ports of the model and the capabilities of the simulator.

The interface that is defined by the standard mainly consists of the following functions:

```
fmiComponent fmiInstantiateSlave(fmiString instanceName,
                                 fmiString fmuGUID,
                                 fmiString fmuLocation,
                                 fmiString mimeType,
                                 gmiReal timeout,
                                 fmiBoolean visible,
                                 fmiBoolean interactive,
                                 fmiCallbackFunctions functions,
                                 fmiBoolean loggingOn);
```

[1] A list is available at `https://www.fmi-standard.org/tools`.

```
fmiStatus fmiInitializeSlave(fmiComponent c,
                             fmiReal tStart,
                             fmiBoolean StopTimeDefined,
                             fmiReal tStop);
fmiStatus fmiTerminateSlave(fmiComponent c);
fmiStatus fmiResetSlave(fmiComponent c);
void fmiFreeSlaveInstance(fmiComponent c);
fmiStatus fmiSetReal (fmiComponent c,
                      const fmiValueReference vr[],
                      size_t nvr,
                      const fmiReal value[]);
fmiStatus fmiSetInteger(fmiComponent c,
                        const fmiValueReference vr[],
                        size_t nvr,
                        const fmiInteger value[]);
fmiStatus fmiSetBoolean(fmiComponent c,
                        const fmiValueReference vr[],
                        size_t nvr,
                        const fmiBoolean value[]);
fmiStatus fmiSetString (fmiComponent c,
                        const fmiValueReference vr[],
                        size_t nvr,
                        const fmiString value[]);
fmiStatus fmiGetReal(fmiComponent c,
                     const fmiValueReference vr[],
                     size_t nvr,
                     fmiReal value[]);
fmiStatus fmiGetInteger(fmiComponent c,
                        const fmiValueReference vr[],
                        size_t nvr,
                        fmiInteger value[]);
fmiStatus fmiGetBoolean(fmiComponent c,
                        const fmiValueReference vr[],
                        size_t nvr,
                        fmiBoolean value[]);
fmiStatus fmiGetString(fmiComponent c,
                       const fmiValueReference vr[],
                       size_t nvr,
                       fmiString value[]);
fmiStatus fmiDoStep(fmiComponent c,
                    fmiReal currentCommunicationPoint,
                    fmiReal communicationStepSize,
                    fmiBoolean newStep);
```

Our approach is to use a generic implementation that loads a given FMU and provides an API that is more idiomatic for the usage inside Java applications. Indeed, within Advance, the multi-simulation masters will be written in Java or Groovy and run on the JVM (usually within hte Eclipse IDE provided by Rodin). A functional mock-up unit is wrapped in a class called FMU. It is created by instantiating the class providing the name of the zip file. This will automatically load the description file, setup the native functions and start the FMU using the `fmiInitializeSlave` function. After loading the native library the user can initialize the FMU providing the simulation duration as specified in the FMI standard, the Java method is similar to the C version. We have decided to make the getter and setter functions more idiomatic. For each of the types that can be read or written we created a getter method (getInt, getBoolean, getDouble and getString) that takes a variable name as its parameter and returns the value at the current point in time. We also provide a set method that takes a variable name and a value and writes the value back to the FMU. Computation steps are executed using the doStep method that calls the `fmiDoStep` function of the FMU. Finally tearing down the FMU is done using the terminate method. The terminate function will not only stop the computation but also free the memory. It combines the `fmiTerminateSlave` and `fmFreeSlaveInstance` functions from the FMI specification. The FMU wrapper has been implemented and is included in the ProB 2.0 integration builds. It uses a modified version of the JFMI wrapper from the Ptolemy project [Thec].

FMI does not impose many restrictions on the master algorithm, the master has to be rewritten from scratch for each system. Using the Groovy scripting language this can be done already in the current implementation. We want to provide the user with a more pleasant experience by supporting a generic master that can be configured from the UI and customized using the scripting language. Such a master needs to know about all the units involved in co-simulation and connections between them. In the FMI parlance this information is called component connection graph and is used to validate assembled units, derive a suitable co-simulation algorithm depending on the topology and other properties of the graph and coordinate co-simulation process.

We are planning to develop a component diagram editor and co-simulation driver as part of the Rodin platform that would enable users to import and instantiate existing FMUs and Event-B machines as components and visually construct a graph model that can be used by the master. Each component on the diagram will have a configurable number of input and outputs ports, compatible with FMI standard types, which can be connected to the corresponding ports of other components via connectors. The diagram will provide controls for running the master and display the simulation state with time, including exchanged signal values between ports and internal state variables of components.

The metamodel of connection graph and the editor will be based on the existing open-source frameworks provided by Eclipse and Rodin and will make use of the Event-B metamodel.

The component diagram tool will support both FMI components, bundled as `.fmu` archives, and Event-B components. An Event-B component may represent a single Event-B machine or a hierarchy of machines with a composed machine as a root element and multiple nested machines that constitute the composition. Each nested machine can be a composed machine itself. We are aiming at supporting such complex compositional structures at the diagram level, so the users will be able to import composed machines and their constituent elements and link them with other components. The interaction between the elements of composed machines may also be part of the visual notation and displayed during the simulation.

6.2 Co-simulation demonstrator

As a demonstrator we implemented a version of a watertank system. For co-simulation the system consists of a FMU written in C that models the continuous part of the system and a controller written in classical B that models the decision computation. The FMU is being loaded using the previously described wrapper class.

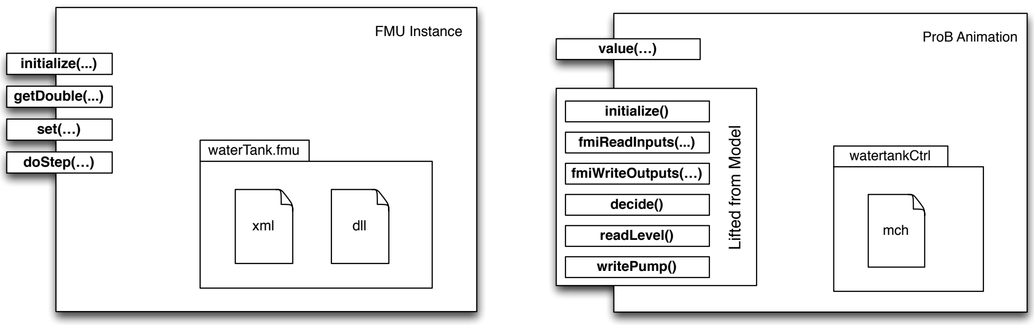


Figure 6.1: FMU and Animation instance in ProB

Figure 6.1 shows the two entities that are used in the co-simulation. On the left we see the waterTank.fmu model wrapped in an instance of the FMU class. The class provides generic methods to access variables, initialize the simulation and perform simulation step. The figure only shows the methods we actually use in the simulation. On the right we see the B model loaded into the ProB animator. The animator provides a generic method to access variables and model-specific methods that were lifted from the model using Groovy meta-programming. For each event of the model ProB creates a method that can be called from Java.

The controller algorithm is shown in figure 6.2. We begin by reading the port values. This step is not part of the decision process, it is a representation of the fact that the current value from the sensor arrives at the port of the controller. Then we read the value from the port into the controller, the controller decides if the pump should be switched on or off and the result is written to the outgoing port of the controller. These three steps that model the controller making a decision do consume some time. Finally the values are written into the port without consuming time.

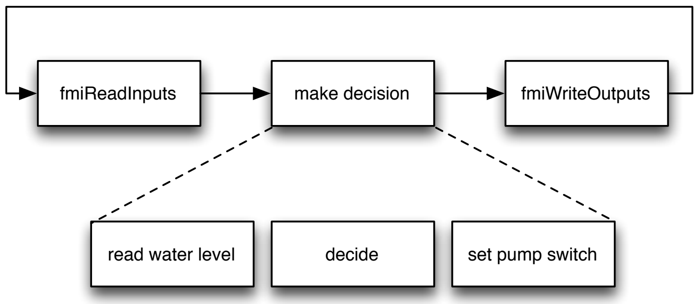


Figure 6.2: Controller algorithm

We developed a master written as a Groovy script shown in figure 6.4 that coordinates the two components and implements the wiring. The outline of the algorithm is shown in figure 6.3. The master initialized both components (1) and then sets the values of the input ports (2). It then executes the decision making process on the controller which will take some time Δt and then simulates the environment for Δt time units (3). The reason is that during the time the computation takes water will continue to flow into or out of the tank. Finally the master retrieves the output of the controller (4) for the next cycle that starts at point (2).

The master script also converts between the types used in the C implementation (double) and the B controller (integer). For the sake of simplicity we decided that we stick to fixed precision values and we multiply each dou-

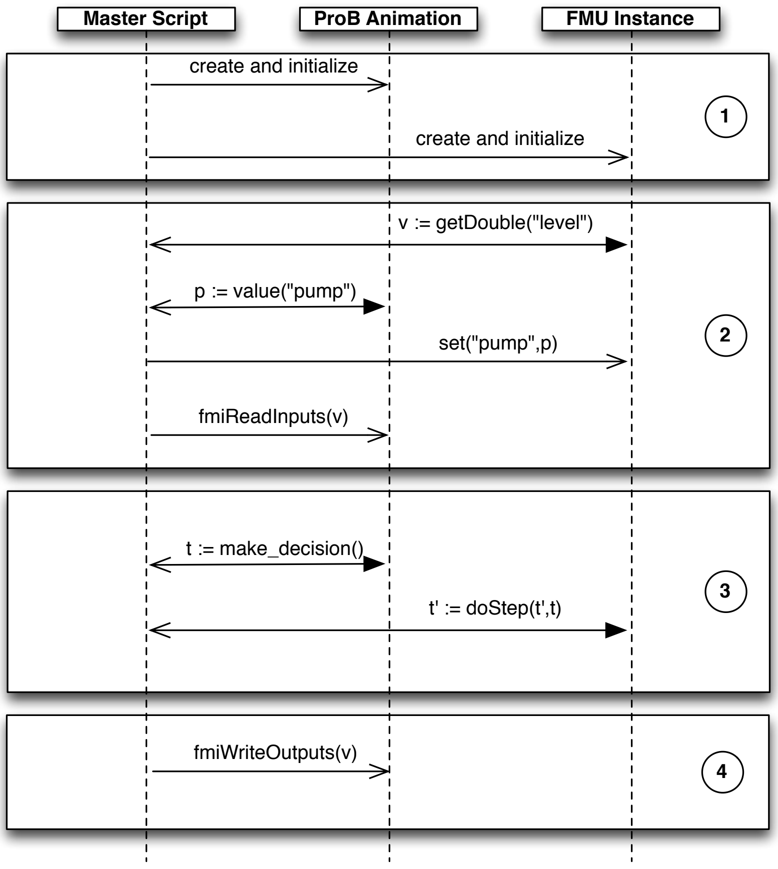


Figure 6.3: Communication between master and slaves

ble by 1000 and cast them to integer. Vice versa we divide each integer from the B controller by 1000.

The output of our co-simulaton demonstrator is shown in figure 6.5. The water tank starts at a level of 1000, the pump is turned off. The controller should establish that the level is between $800 - 2\epsilon_{out}$ and $2000 + 2\epsilon_{in}$ where ϵ_{out} and ϵ_{in} are the maximal water flow rates.

```
import de.prob.cosimulation.FMU

master = { dir ->
  ctrl = api.b_load(dir+"/waterTankCtrl.mch") as History
  ctrl = ctrl.anyEvent(); // setup constants
  ctrl = ctrl.anyEvent(); // initialize machine

  fmi = new FMU(dir+"/waterTankEnv.fmu")
  fmi.initialize(0.0, 10.0)

  time = 0.0

  while(time < 10) {
     // Transfer initial values to ports
     level = fmi.getDouble("level")
     pump  = ctrl.getCurrentState().value("fmiPump")=="TRUE";
     blevel = (level*1000) as int
     fmi.set("pump",pump)

     println "Setting inputs. pump="+pump+" level="+blevel

     ctrl = ctrl.fmiReadInputs("l="+blevel)

     // Let controller decide what action it should perform
     println "Controller decides"
     ctrl = ctrl.readLevel().decide().writePump()

     ctrltime = (ctrl.getCurrentState().value("time") as int)/1000

     // Read time from Controller and simulate environment
     // Side effect: new value of level is available
     time = fmi.doStep(time,ctrltime-time);

     // Store Controller's decision on wire
     ctrl = ctrl.fmiWriteOutputs()
  }
}
```

Figure 6.4: Groovy code of the FMI master

```
time=0.0 pump=false level=1000
time=0.14 pump=false level=761
time=0.28 pump=true level=481
time=0.42 pump=true level=579
time=0.56 pump=true level=719
time=0.7 pump=true level=859
time=0.84 pump=true level=1000
time=0.98 pump=true level=1140
time=1.12 pump=true level=1280
time=1.26 pump=true level=1420
time=1.4 pump=true level=1560
time=1.54 pump=true level=1700
time=1.68 pump=true level=1840
time=1.82 pump=true level=1980
time=1.96 pump=true level=2119
time=2.1 pump=false level=2260
time=2.24 pump=false level=2022
time=2.38 pump=false level=1742
time=2.52 pump=false level=1462
time=2.66 pump=false level=1182
time=2.8 pump=false level=902
time=2.94 pump=false level=622
time=3.08 pump=true level=342
time=3.22 pump=true level=440
time=3.36 pump=true level=580
time=3.5 pump=true level=720
time=3.64 pump=true level=860
time=3.78 pump=true level=1000
time=3.92 pump=true level=1140
time=4.06 pump=true level=1280
time=4.2 pump=true level=1420
time=4.34 pump=true level=1560
time=4.48 pump=true level=1700
time=4.62 pump=true level=1840
time=4.76 pump=true level=1980
time=4.9 pump=true level=2120
```

Figure 6.5: Output of the co-simulation run

6.3 Plan

A first prototype for FMI-based multi-simulation has actually already been implemented (as described above). The plan for the coming year is as follows:

- Within the next months, by June 2013, the ProB-driven console based multi-simulation will be available first internally to project participants and then later in 2013 in the official release of ProB.
- a prototype of editor for Component wiring and simple generic master will be developed alongside, and will be ready by late summer 2013.
- By fall 2013 we expect feedback from the industrial case studies, which will enable us to improve the generic simulation master. In particular, we will provide support for coordinating timed and continuous components (without having to hard-code these aspects into the master).

Chapter 7

Model Testing

The scripting approach can also be useful for model testing and testcase generation. For model testing, we can save and replay execution traces. Saving a trace is done by generating a groovy script and replaying is done by executing that script. In contrast to previous approaches [Elb09] this scripting approach is not limited to simple linear traces because we have the full expressivity of the host language at our hands.

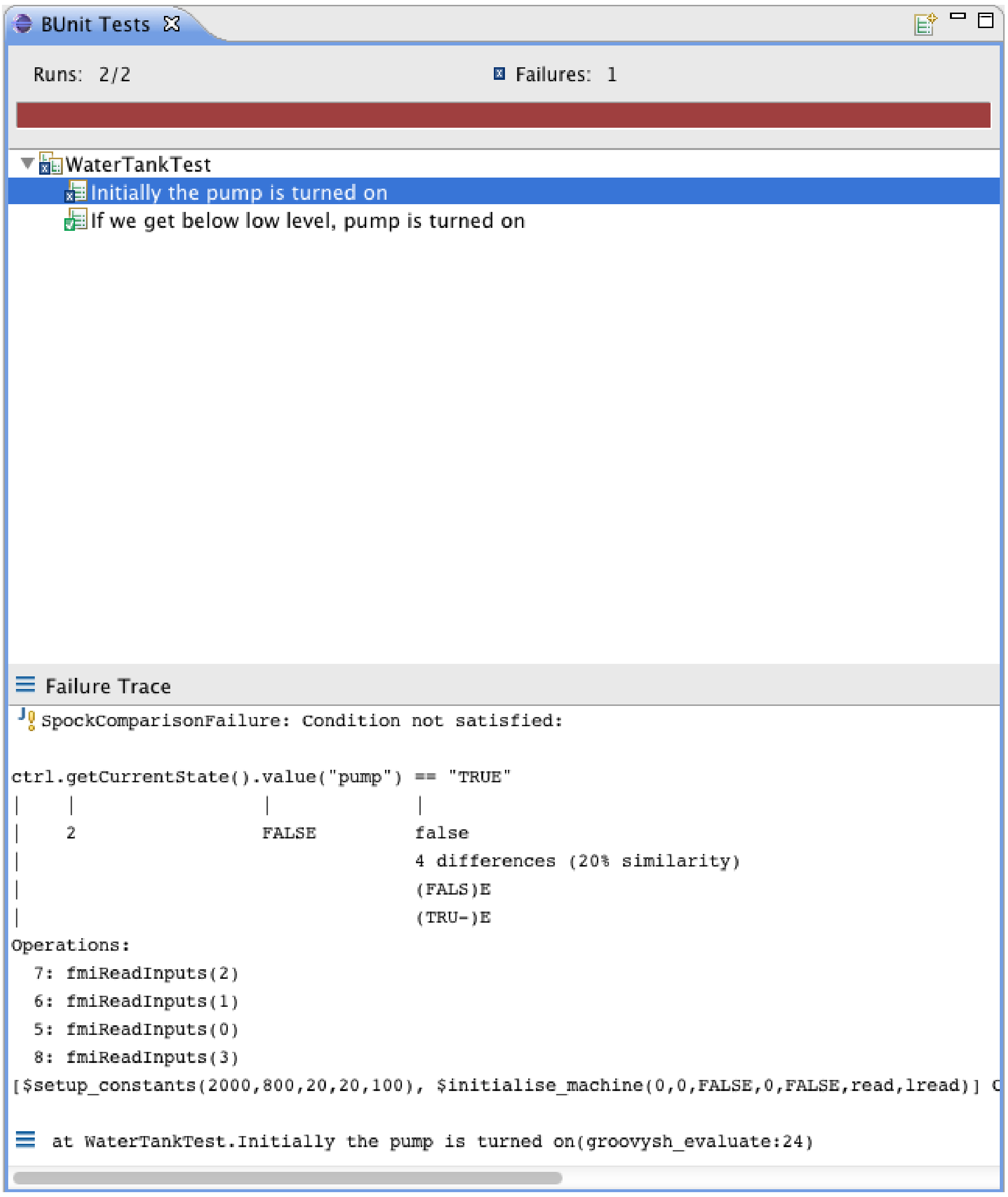
The new ProB plug-in has a built in JUnit4 test runner. There is also built in support for the Spock test framework[1]. This means that users can write JUnit4 or Spock tests using the ProB abstractions. Tests are written within groovy scripts. In order to use the BUnit test runner, the user has to write a test class that is compatible with JUnit4 or Spock and then create an new instance of the test class in the last line of the script. Then, when the script is run, the groovy engine will recognize the class and run it through the test runner. An example of a model test for the watertank example from chapter 6 written as a Spock specification could look like this

```
import spock.lang.Specification
import de.prob.scripting.Api

class WaterTankTest extends Specification {
    static Api api
    def ctrl

    def setup() {
      ctrl = api.b_load("waterTankCtrl.mch") as History
      ctrl = ctrl.anyEvent(); // setup constants
      ctrl = ctrl.anyEvent(); // initialize machine
```

[1]http://www.spockframework.org

```
    }

    def "Initially the pump is turned on"() {
      expect:
       // This test will fail! The value is FALSE
       assert ctrl.getCurrentState().value("pump") == "TRUE"
    }

    def "If we get below low level, pump is turned on"() {
      when:
       ctrl = ctrl.fmiReadInputs("l=799") // set level < Low
       ctrl = ctrl.readLevel().decide().writePump() // decide
      then:
       assert ctrl.getCurrentState().value("pump") == "TRUE"
    }

}

WaterTankTest.api = api
new WaterTankTest()
```

When the test cases have been created, the user can use the BUnit Test View in the ProB plug-in to select the directory that contains the test cases. Then all tests within the directory will be run. The view will show whether a test succeeded, failed, or produced an error (see Fig. 7.1). When an error or a failure is produced, a stacktrace or explanation with the desired information is also displayed. Using the specification for BUnit tests, it should not be difficult for users to generate test cases.



BUnit Tests

Runs: 2/2 Failures: 1

WaterTankTest
Initially the pump is turned on
If we get below low level, pump is turned on

Failure Trace

```
SpockComparisonFailure: Condition not satisfied:

ctrl.getCurrentState().value("pump") == "TRUE"
|    |                 |           |
|    2                 FALSE       false
|                                  4 differences (20% similarity)
|                                  (FALS)E
|                                  (TRU-)E
Operations:
  7: fmiReadInputs(2)
  6: fmiReadInputs(1)
  5: fmiReadInputs(0)
  8: fmiReadInputs(3)
[$setup_constants(2000,800,20,20,100), $initialise_machine(0,0,FALSE,0,FALSE,read,lread)] C

  at WaterTankTest.Initially the pump is turned on(groovysh_evaluate:24)
```

Figure 7.1: Model Testing UI

Chapter 8

Simulation using Tasking Event-B's Generated Code

8.1 Abstract

One way to produce FMUs from a B component is via code generation: instead of simulating a B component we generate code from it and use that code within an FMU unit. The advantage of this is obviously performance.

The RODIN, and DEPLOY projects have laid solid foundations for further theoretical, and practical (methodological and tooling) advances with Event-B; we investigated code generation for embedded, multi-tasking systems. This work describes activities from a follow-on project, ADVANCE; where our interest is co-simulation of cyber-physical systems. We are working to better understand the issues arising in a development when modelling with Event-B, and animating with ProB, in tandem with a multi-simulation strategy. With multi-simulation we aim to simulate various features of the environment separately, in order to exercise the deployable code. This work has two main contributions, the first is the extension of the code generation work of DEPLOY, where we add the ability to generate code from Event-B state-machine diagrams. The second describes how we may use code, generated from state-machines, to simulate the environment, and simulate concurrently executing state-machines, in a single task. We show how we can instrument the code to guide the simulation, by controlling the relative rate that non-deterministic transitions are traversed in the simulation.

8.2 Introduction

Event-B [Abr10], and supporting tools have been developed during the RODIN and DEPLOY [Theb] projects, and continues in ADVANCE [Thea]. Some industrial partners were interested in the formal development of multi-tasking, embedded control systems. We developed an approach for automatic code generation, from Event-B models, for these type of systems [EB11]. Event-B uses set-theory, predicate logic and refinement to model discrete systems. The basic structural elements of Event-B models are contexts and machines. Contexts describe the static aspects of a system, using sets, constants, and axioms. The contents of a Context can be made visible to a machine. Machines describe the dynamic aspects of a system, in the form of state variables, and guarded events, which update state. Required properties are specified using the invariants clause. The invariants give rise to proof obligations.

State-machine diagrams [Eve] can be added to a machine. Each contains an initial state, typically contains one or more transitions, one or more other states, and possibly a final state. A transition 'elaborates' one or more events; that is, a transition describes the atomic state updates that occur during the change from one state to the next. We use an example of an automotive engine stop-start controller, loosely based on [GLG$^+$11], to illustrate our approach. The system aims to save fuel by switching the engine off when the car is stationary. Fig. 8.1 is an example of a state-machine diagram, *EngMode*. Initially the state-machine is in the *ENG_ OFF* state, and may go the *ENG_ CRANKING* state via transitions *s1* or *userStart*, and so on. In the properties we define 'translation type' as *Enumeration*. The underlying Event-B model, uses a set-partition of the states, as shown below. The current state of the state-machine is recorded in a variable $EngMode \in EngMode\_STATES$, where $EngMode\_STATES$ is a partition of the states of the EngMode state-machine,

$$\begin{aligned} &partition(EngMode\_STATES, \{ENG\_STOPPING\}, \\ &\qquad \{ENG\_CRANKING\}, \{ENG\_RUNNING\}, \{ENG\_OFF\}) \end{aligned} \tag{8.1}$$

In this paper we describe how we extend the code generation work of DEPLOY; we add the ability to generate code from Event-B state-machine diagrams. We then describe how we may use code, generated from state-machines, to simulate the environment, and simulate concurrently executing state-machines, in a single task. We describe how we guide the simulation, that is, control the relative rate that non-deterministic transitions are traversed, using additional guards on the transition implementations.

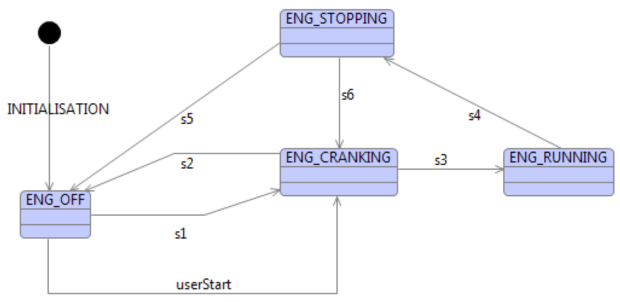


Figure 8.1: EngMode State-machine

```
TaskBody ::=
  TaskBody ; TaskBody
  | if EventName
    (elseif EventName)*
     else EventName                EventName ::= String
  | while EventName
  | output String VariableName     VariableName ::= String
```

Figure 8.2: Task Body Syntax

8.3 Tasking Event-B

Tasking Event-B [EB11, ERB12] is an extension to the Event-B; where Event-B elements are restricted to implementable types. If required we use decomposition [SPHB10, SB10] to separate the system into sub-components. At an appropriate stage we introduce implementation specific constructs to guide code generation. These constructs are underpinned by Event-B operational semantics; Tasking Event-B introduces three main constructs:- AutoTask, Environ, and Shared Machines. AutoTask Machines model controller tasks (in the implementation). Environ Machines model the environment, and Shared Machines provide a protected resource for sharing data between tasks.

Tasks bodies are specified using the syntax shown in Fig. 8.2. We can use (;) sequence , (if-elsif-else) branching, (do) looping, and text output to the console.

8.4 Translation of a Task Body

To simplify the discussion, our example uses a single tasking approach. We will not consider here the issue of multi-tasking. We therefore need only to give a brief overview of AutoTask Machine translation, since it will not be synchronized with a Shared Machine. Given an event $E \triangleq g \rightarrow a$, we map action a to a program statement $a\prime$, and guard g to a condition $g\prime$, if g

exists. The guard should be $\top$ for events used in sequences, but may be any implementable predicate for use in branching and looping statements. An example translation of branching follows, where events $e_1 \triangleq g1 \rightarrow a_1$ and $e_2 \triangleq g_2 \rightarrow a_2$, are used in the task body,

$$\begin{gathered}\textbf{if } \; e_1 \textbf{ else } e_2 \textbf{ endif} \\ \rightsquigarrow \\ \textbf{if } g\prime_1 \textbf{ then } a\prime_1 \textbf{ else } a\prime_2 \textbf{ end if};\end{gathered}$$

The branching construct of the task body contains events e_1 and e_2, and translates to a branching construct in the program code. The guard $g\prime_2$ does not appear in the code, but a proof obligation can be generated to ensure that $g_2 = \neg g_1$. The code generator could be augmented to automatically generate proof obligations to show that branch guards are disjoint and complete.

8.5 The Automotive Stop-Start Model

A typical approach to multi-tasking in hybrid systems, relies on a *write-read-process* protocol. The shared variable store, shown in Fig. 8.3, is used by the various modules; to write to, then read from. In such a system, each task keeps a local copy of the parts of the state that it needs to deal with. In the *write-read-process* protocol, all tasks write to the store, all tasks then read from the store. Only when all tasks have updated their local copies of shared state, can processing take place. The task iterates these steps in a loop. In our tool we simulate the concurrent implementation using *sequential* code generated from a single AutoTask Machine. The deployable modules of Fig. 8.3 can be implemented in a multi-tasking environment if the execution order of the protocol is preserved.

In our sequential simulation, we use a single AutoTask Machine, which contains both controller and environment state-machines; and define write and read behaviour in the machine's task-body construct. We have already seen the Stop-Start (SSE) system's *EngMode* state-machine, in Fig. 8.1. In addition to this we have Clutch, Gear and Steering environment state-machines. There are three controller modules, the SSE Module which decides whether to issue stop or start commands based on the engine state. It receives output from the HMI Controls module. HMI Controls monitors the clutch, gear, and steering controls to see if automatic stop or start should be enabled.

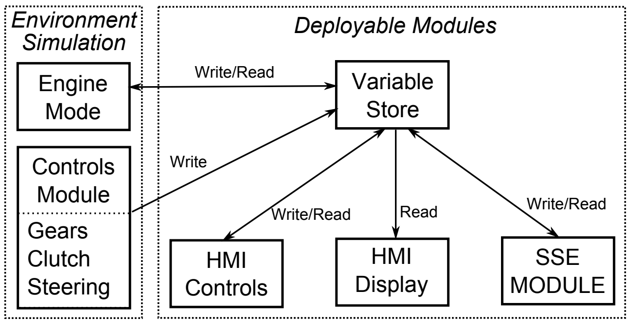


Figure 8.3: Overview of the Stop-Start Architecture

8.6 The IO Architecture

We have defined the state-machines of the system and we can now specify the IO between the modules via the shared variable store. The store contains a copy of all of the variables involved in IO between modules. Each of the modules may send data and receive data from the variable store. If we take, as an example, the engine's IO, we output the engine state and speed to the shared variable store. All variables in the store are prefixed 'STO_', and variables in the engine module (other than state names) are prefixed 'ENG_', so the following event updates the shared variable store's copies of the engine state. Each state-machine has a send (*write*) and receive (*read*) event which has the state-machine name and *send* or *recv* as a suffix.

$$\begin{aligned} Eng\_send \triangleq\ & STO\_EngMode := EngMode \\ & \parallel STO\_EngineSpeed := ENG\_EngineSpeed \end{aligned}$$

8.6.1 Modelling Starting and Stopping the Engine

The *EngMode* state-machine keeps track of the engine mode, i.e. off, running, cranking, or stopping. The engine is initially in the *ENG_OFF* state. We model the ultimate task of the SSE system, the automatic engine start, with the *s1* event. This is enabled after receiving an engine start order from the Stop-Start Controller module (the SSE Module's SSEMode state-machine, introduced later). The *s1* event follows,

```
$s1 \triangleq$ when $EngMode = ENG\_OFF \wedge ENG\_Start\_Order = TRUE$
    then $EngMode := ENG\_CRANKING$
    end
```

The predicate and action involving *EngMode* are automatically generated in the translation from the state-machine diagram. The guard,

$$ENG\_Start\_Order = TRUE$$

is added by the developer to indicate that the engine should enter the cranking state when a Start Order has been received. The engine may also be started manually, as modelled by the userStart event. When the engine is running at a sufficient rate $s3$ sets the engine state to *ENG_ RUNNING*,

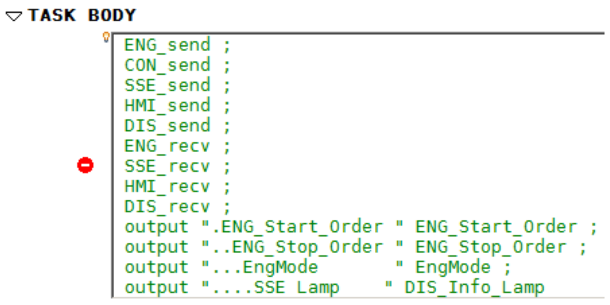
```
$s3 \triangleq$ **when** $EngMode = ENG\_CRANKING$
            $\wedge\ Eng\_EngineSpeed >= Eng\_Idle\_Speed$
    **then** $EngMode := ENG\_RUNNING$
    **end**
```

When the engine is running it can be stopped automatically by the SSE module. The *HMI_ Controls* module checks to see if it is in neutral gear, steering not-used, and clutch released. If it is, *HMI_ Stop_ EnaT* sets *HMI_ Stop_ Ena* to true. This eventually gets passed to the *SSEMode module* via the shared store.

```
$HMI\_Stop\_EnaT \triangleq$
**when** $HMI\_Gear = NEUTRAL \wedge HMI\_Steer = NOT\_USED$
 $\wedge\ HMI\_Clutch = RELEASED$
 $\wedge\ HMI\_ControlsSM = HMI\_OPERATION$
**then** $HMI\_Stop\_Ena := TRUE \parallel HMI\_Strt\_Req := FALSE$
**end**
```

Event $t7$, of the SSE Module, updates its SSE_Stop_Order and SSE_Start_Order if SSE_Stop_Ena, and the other guards, are satisfied. It is then copied to the variable store, and subsequently read by the engine module.

```
$t7 \triangleq$
**when** $SSEMode = SSE\_OPERATION \wedge SSE\_Stop\_Req = TRUE$
 $\wedge\ SSE\_EngMode = ENG\_RUNNING \wedge SSE\_Stop\_Ena = TRUE$
**then** $SSEMode := SSE\_STOPPING \parallel SSE\_Stop\_Order := TRUE$
 $\parallel SSE\_Start\_Order := FALSE$
**end**
```



```
TASK BODY
ENG_send ;
CON_send ;
SSE_send ;
HMI_send ;
DIS_send ;
ENG_recv ;
SSE_recv ;
HMI_recv ;
DIS_recv ;
output ".ENG_Start_Order " ENG_Start_Order ;
output "..ENG_Stop_Order " ENG_Stop_Order ;
output "...EngMode         " EngMode ;
output "....SSE Lamp     " DIS_Info_Lamp
```

Figure 8.4: The Task Body Specification

8.7 The Task Body

We specify the sequence of events in the Task Body in the 'usual' Tasking Event-B style, seen in Fig. 8.4. We have specified that send events occur before the read events. This is necessary to ensure the latest state is made available for the state-machine evaluation. The Task Body is periodic, and generates a loop in the implementation. The order of processing is as follows: 1) Initialisation of state. 2) Evaluate state-machines. 3) Send updated values to the variable store. 4) Read updated values from the variable store; then go to 2, and repeat. The sequence {3,4,2}, in the task body, corresponds to the *write-read-process* protocol, which follows initialisation. Fig. 8.4 also shows the *output* clause, for text output to the console. The next section provides details of the translation to Ada code.

8.8 Translating State-Machines to Code

To illustrate the translation process we show a pseudo-code implementation in the Ada style. We have seen how state-machine states are modelled by an enumeration partition, and we use this in the implementation. The partition of Equation 8.1 is translated to the following pseudo-code.

```
**type** $EngMode\_STATES\ = ($
    $ENG\_STOPPING,\ ENG\_CRANKING,$
    $ENG\_RUNNING,\ ENG\_OFF); \ldots$
```

We store the global constants and types, where the type *EngMode_ STATES* is an enumeration of the state-machine states. Recall also, that we generate a state variable $EngMode$ which is typed as $EngMode \in EngMode\_STATES$, to keep track of the state; it has the initial value *Eng_ OFF*. We use the diagram and the initialisation event to generate the following code:

$$EngMode : EngMode\_STATES := ENG\_OFF;$$

The main program invokes the state-machine implementations in a loop, once per cycle. Each state-machine diagram maps to a procedure. State-machine procedures are called exactly once before the sends to, and reads, from the variable store. The evaluation of each state-machine procedure is independent of the others state-machines, since each keeps a local copy of the state, copied from the variable store. Each state-machine procedure has a state variable v, states s_i, and implemented actions a_i. To each state-machine procedure, we add to a *case* statement,

```
**case** $v$ **when** $s_1 => a_1$;
**when** $s_2 => a_2$; ...
**when** $s_n => do - nothing$;
```

Translation of our example gives rise to the following pseudo-code,

```
procedure $EngModestateMachine$
case $EngMode$
  when $ENG\_STOPPING =>$
  if $((ENG\_EngineSpeed = 0))$ then
    $EngMode := ENG\_OFF;$
  elsif $((ENG\_Start\_Order = true))$ then
    $EngMode := ENG\_CRANKING;$
  else $do - nothing;$
  end if;
  when $ENG\_CRANKING =>$
  if $((ENG\_EngineSpeed = 0))$ then
    $EngMode := ENG\_OFF;$
  elsif $((ENG\_EngineSpeed >= Eng\_Idle\_Speed))$ then
    $EngMode := ENG\_RUNNING;$
  else $do - nothing;$
  end if;
  when $ENG\_RUNNING => \ldots$
end case;
```

We can see that each of the case's *when* statements contains a branching statement. This is because each state of the state-machine has at least two branches; a do-nothing transition, plus one or more outgoing transitions. The do-nothing transition is not explicitly shown on the diagram. A do-nothing transition can be added to each state, since adding a *skip* event is a valid refinement. It is implemented by the **else do-nothing;** branch. Other branches are translated from states with more than one outgoing transition. This may be seen in the *ENG_ STOPPING* case in the example. The branch conditions are mapped from the guards of the events ($s5$ and $s6$) that elaborate the outgoing transitions.

8.9 Manipulating State Machine Transitions

The generated code from our example is compiled to an executable file and run. In essence we have generated implementable code for the controller state-machines, and a simulation of the environment from the environment state-machines. When we run the code we find that most of the state remains unexplored, and this is due to the non-determinism in the state-machines. This section identifies how we can guide a simulation, by reducing the non-determinism in the state-machines.

For the controller state-machines, each state's outgoing transitions are disjoint and complete; in other words, a transition is always taken in the simulation. However, in the environment, it is unlikely that the clutch changes state so frequently. We do have the implicit *do-nothing* transition on environment state-machine states, but we need this to happen more often than the other transitions. We must have some control over the relative rate that non-deterministic transitions are traversed in the simulation. As it stands, any outgoing transition is equally likely to occur. To solve this in the simulation, we introduce an enabling variable $q \in 0 \mathrel{..} n$ and a random variable $r \in 0 \mathrel{..} n$, and use the random variable in a case-statement's branch conditions. Variable q is calculated once at the beginning of the simulation, but a new random variable r is calculated at each state-machine evaluation. The event $g \mapsto a$ in Event-B terms is implemented as a branch $g \wedge r = q \mapsto a$ in a case-statement.

We now suggest how we may generate, and use the variables q and r in simulation. This aspect is work in progress, but we believe the approach will be useful for generating test scenarios, and therefore will help to achieve full test coverage. By adding a guard to the branch condition we can influence the path taken through the code during simulation. In effect, we reduce the non-determinism in the state-machines, which allows us to guide the simulation, and therefore the exploration of the state-space.

One question is, how to choose a value of n? We could base it on the total number of outgoing transitions of the state involved, but this would not give a large enough value. A typical state may have four transitions so a random number $r \in 0 \mathrel{..} 4$ could be used. However, we wish to manipulate the probability of a branch being taken, so that a branch is very unlikely to be taken; therefore, a much larger value for n is required. So, we calculate n based on the number of tests that would be required, for test coverage of all transitions, in all states. Likewise, the value of q must be unique within the case-statement; we just allocate an arbitrary, but unique value, close to n. In future work we will investigate how we could modify n during simulation runs, and use this value to reduce the probability of a simulation traversing previously explored state. In the code fragment below, we add the probabilistic condition to the branch of the case-statement, where $r = StartStop01b\_random$ ($StartStop01b\_random$ is a random variable in the implementation code) and $q = 3990$.

```
case $EngMode$ is
when $ENG\_STOPPING$ =>
if $((ENG\_EngineSpeed = 0))$ and $(StopStart01b\_random = 3990)$ then
    $EngMode := ENG\_OFF; \ldots$
```

Adding the branch condition gives us control over the likelihood that a particular transition from a state will be taken when the state-machine is evaluated. We manually modify the conditions, to affect the behaviour of the simulation. We may wish to focus on exploring the state in a particular region. For instance, to test an engine-stop scenario, we require that the engine is in the ENG_ RUNNING state, the gear is in NEUTRAL, the clutch is in the RELEASED state, and the steering NOT_USED. Fig. 8.5 shows that we want large probabilities of transitions leading to the states that we want, and small ones departing.

For a given simulation run we can define *attracting* and *repelling* states. Here, ENG_RUNNING is an attracting state; that is, we want the state-machine to be in that state or moving towards it most of the time. To achieve this we can adjust the branch conditions, to increase the probability of the transitions that lead to that state, being taken. For instance to increase the probability of the engine going from ENG_OFF to ENG_CRANKING we can modify the statement to read (StopStart01b_random <= 3990). In addition to this, we propose to record the navigated transitions, for transition coverage analysis. So, we will be able to use the data also, to guide the simulation. We show two simulation runs here, with the text output defined in the Task Body, *Run1* uses the 'unmodified', generated code; it simply

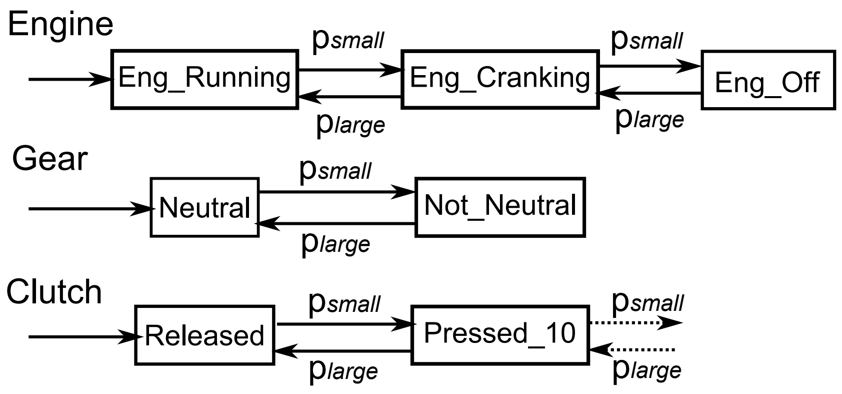


Figure 8.5: Controlling the Simulation

loops and never reaches the ENG_RUNNING state. With the branch conditions modified, as described, *Run 2* shows the simulation cycling from *ENG_RUNNING* to *ENG_OFF*; and with the indicator lamp changing to inform the driver of the situation.

Run 1
.ENG_Start_Order FALSE
..ENG_Stop_Order FALSE
...EngMode ENG_OFF
....SSE Lamp OFF . . .

Run 2
.ENG_Start_Order FALSE
..ENG_Stop_Order TRUE
...EngMode ENG_RUNNING
....SSE Lamp OFF
.ENG_Start_Order FALSE
..ENG_Stop_Order TRUE
...EngMode ENG_STOPPING
....SSE Lamp ORANGE_STOP
.ENG_Start_Order FALSE
..ENG_Stop_Order TRUE
...EngMode ENG_OFF . . .

8.10 Conclusions

We have shown how we generate code from State-machines, and illustrated the approach with a case study based on an automotive engine controller, automatic stop-start system. We describe how we simulate the environment, and a multi-tasking implementation. We gain an insight into how we adjust the conditions to provide meaningful simulation runs. In future work we intend to record the transition coverage, and feed this back to the simulator, to ensure all transitions are covered. We will also investigate the interaction between the generated code, environment simulations, and ProB.

Chapter 9

Outlook on Continuous Time and Hybrid Systems

Although the current focus of research is on cycle-based and discrete time systems, we have started to investigate the implications of continuous and hybrid modelling for ADVANCE.

We have carried out a range of modelling and simulation experiments with Modelica language [A+05] to evaluate its applicability to proposed idea of co-simulation framework with Event-B. One of the advantages of Modelica is the ability to mix natural description of continuous-time physical processes, defined by differential-algebraic equations, with discrete-time behaviour, specified by *when* clause, which can be periodic, time- or event-triggered. This aspect in particular has captured our attention as a possible linking point between discrete-time simulation and its counterpart in Event-B. In addition, openness and large support by simulation tool community suggested Modelica as a suitable technology.

9.1 Modelling a Water Tank

For our simulation experiment we took a classical example of a water tank that must maintain its water level within two bounds. There are many variations of this system in the literature. One of the most common versions consists of a water tank that has a constant water outflow (leakage) rate v_2, and a water pump that can be turned on or off. When the pump is on it pours the water in the tank with a constant rate v_1. Water tank has some capacity and two water level thresholds $0 < L < H$. The goal of the system is to keep the water level within the interval $[L, M]$. The dynamics of the water level is described by the following equations:

$$\varphi_{in} = \begin{cases} v_1 & \text{if pump is on} \\ 0 & \text{if pump is off} \end{cases} \tag{9.1}$$

$$\varphi_{out} = v_2 \tag{9.2}$$

$$\dot{level}(t) = \varphi_{in} - \varphi_{out} \tag{9.3}$$

where φ_{in} is the incoming water flow rate from the pump and φ_{out} is the outgoing water flow rate, caused by leakage. Schema of the water tank is shown below.

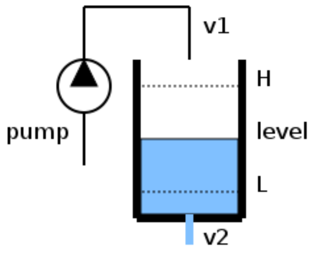


Figure 9.1: Water tank model

As a hybrid system this example has the following components: water pump that represents the plant; water tank with water flow as the environment (and plant); sensors that allow to monitor the water level; and controller that controls the pump. Water pump has two modes of operation: *on* and *off*. In addition, it has activation delay t_{act} s, which defines the time between it being switched on or off and the actual result in terms of changes in the water flow. Water level is described by means of sensors, which have sensing period t_{sen} s. The controller follows a simple strategy: when the pump is off and water level drops to lower bound, the controller turns the pump on; when the pump is on and water reaches upper bound, the controller turns it off again; in between the bounds same mode (on or off) is preserved by controller.

For implementation of controller we adopted a predictive strategy, where control decision is based on predicted water level at the next reading of the sensor, taking into account activation time of the pump [SAZ12]. Supervisory controller has four modes that correspond to pump status: two modes represent the physical pump state (*PumpOn* and *PumpOff*) and the other two define control states (*ControlOn* and *ControlOff*), where the pump is switched on or off, but not yet activated/deactivated. Two-level controller is described by the following hybrid automaton:

The control automaton in Figure 9.2 has several important aspects. States *PumpOff* and *ControlOff* have domain restriction (constraint $level \geq 0$),

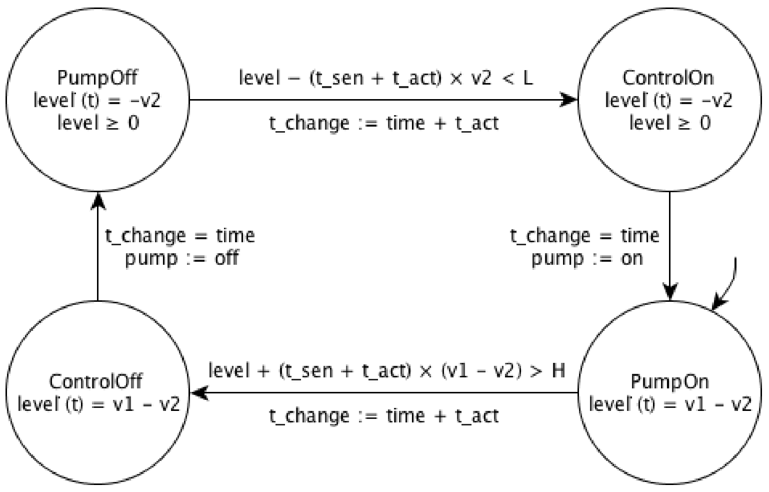


Figure 9.2: Hybrid automaton of the water tank controller

which is necessary to guarantee that water level never gets negative. State transitions are guarded by predicted water level signal from the sensor. In addition, the control is using time variable t_{change} to model the reaction of the pump, i.e. activation delay. Thus, for instance, when the pump is off and water level is decreasing (state *PumpOff*) the controller can predict the next water level reading from the sensor: $level - t_{sen} \times v_2$. Due to activation delay of the pump this prediction needs to be adjusted by $t_{act} \times v_2$. When predicted level tells controller that the lower threshold L will be crossed (guard $level(t_{sen} + t_{act}) \times v_2 < L$ is satisfied), the controller switches to state *ControlOn*, setting time variable t_{change} to time when the pump will actually be activated ($t_{change} := time + t_{act}$). After activation time elapses ($t_{change} = time$) physical pump state is switched to *on* ($pump := on$) and controller moves to state *PumpOn*.

The following code models the system. Notice that in this case state machine is modelled by two discrete variables: *control_state* and *pump_state*. State logic is not using the *algorithm* block anymore, but instead is done within a periodic *when* block that is also modelling the sensor reading. As an outcome, this makes the controller and the sensor synchronised, which was not originally in the specification. Other *when* blocks are used to model the delay, as there are no language constructs for this purpose in Modelica.

```
class WT3
  parameter Real H = 14;
  parameter Real L = 1;
  parameter Real v1 = 3;
  parameter Real v2 = 2;
  parameter Real t_sen = 5;
  parameter Real t_act = 6;
  Real level(start = 1);
  discrete Real detected_level;
  Boolean pump(start = true);
  Boolean control_state(start = true);
  Boolean pump_state(start = true);
  discrete Real t_change;
  annotation(experiment(StartTime = 0.0, StopTime = 30.0, Tolerance = 1e-06));
equation
  // water tank level dynamics
  der(level) = if pump then v1-v2 elseif level > 0 then -v2 else 0;

  // sampling: sensor and controller
  when sample(0, t_sen) then
    detected_level = level;
    control_state = not pre(pump_state) and level - (t_sen + t_act) * v2 < L
      or not level + (t_sen + t_act) * (v1 - v2) > H and pre(pump_state);
  end when;

  // setting activation timer
  when change(control_state) then
    t_change = time + t_act;
  end when;

  // activating pump
  when time >= pre(t_change) then
    pump = control_state;
    pump_state = control_state;
  end when;
end WT3;
```

The simulation plot is shown in Figure 9.3, where both continuous-time (physical) and discrete-time (sensed) signal of the water level is displayed.

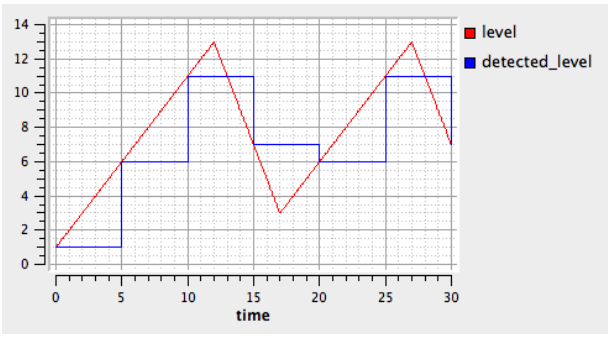


Figure 9.3: Water tank simulation plot: $L = 1, H = 14, v_1 = 3, v_2 = 2,$ $t_{sen} = 5, t_{act} = 2$

9.2 Summary

A water tank has been modelled using the Modelica hybrid modelling capability using a predictive control approach with promising results. In particular, we have shown the potential for using the Modelica *when* clause as a possible linking point between discrete-time simulation and its counterpart in Event-B.

Chapter 10

Improvements to ProB and BMotion Studio based on Requirements from Case Studies

In this chapter we present a few developments that were undertaken in response to requirements coming from the case studies.

10.1 BMotion Studio Developments

10.1.1 Introduction

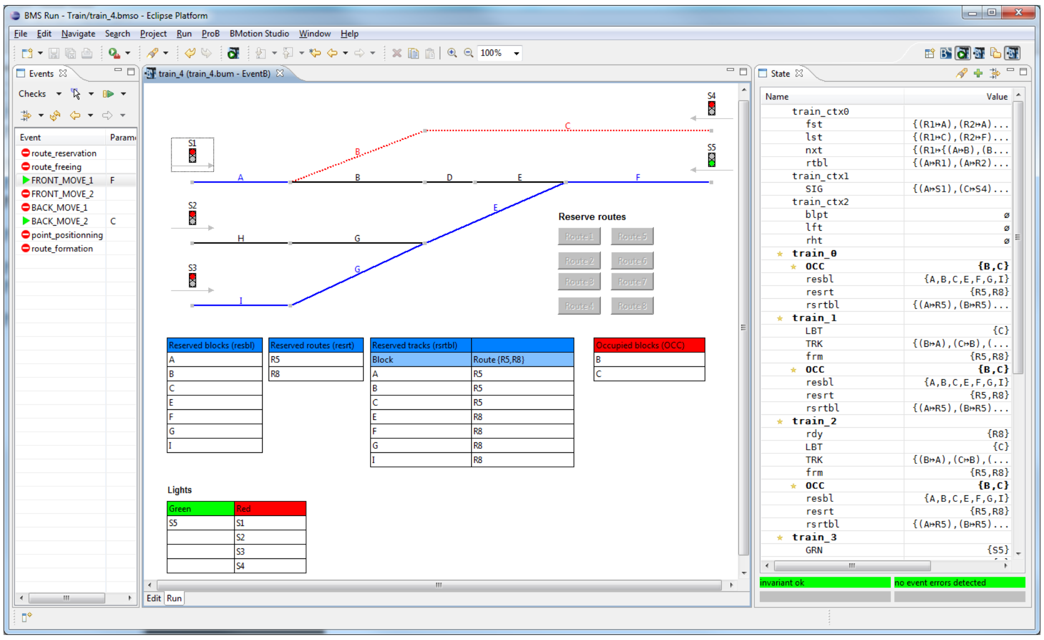
An object of the ADVANCE project is to develop new visualization techniques to aid humans understand of large-scale simulations. The ProB animator [LB03] already allows to check the presence of desired functionality and to inspect the behaviour of a specification by “clicking through” the states of the specification. Several views like the state view, which displays values for variables and constants of the current state supports humans understand of simulations. However, ProB requires still knowledge about the mathematical notation to understand the meaning of a specific state. In particular, large-scale simulations becomes quickly unclear and sometimes a developer and a domain expert desires a different point of view of the model. To overcome this problem, it is useful to create domain specific visualisations.

BMotion Studio [LBL09] is a visual editor which enables the developer of a formal model to set-up easily a domain specific visualisation. BMotion Studio comes with a graphical editor that allows to create a visualisation within the modeling environment. Also, it does not require to use a different notation for gluing the state and its visualisation. Although, BMotion Studio comes with a number of default visualization elements that are sufficient for most visualisations, it can be extended for specific domains.

10.1.2 Table view control

Since, large-scale simulations may become large and the amount of information that will be available for presentation to the user increases, it is important to obtain a meaningful subset of the information and to display it in a way to aid humans understand of the simulation.

In the ADVANCE project we developed a new control for displaying information of the simulation using tables. The table control comes in form of an add-on for BMotion Studio and is fully customizable, which means that the user is able to create individual tables to display subsets of each machine's state. As a consequence, the table control left the choice of what to display at any time to the user.



BMS Run - Train/train_4.bmso - Eclipse Platform
File Edit Navigate Search Project Run ProB BMotion Studio Window Help
Events
Checks
train_4 (train_4.bum - EventB)
State
S1
S2
S3
S4
S5
A
B
C
D
E
F
G
H
I
Reserve routes
Route 1
Route 2
Route 3
Route 4
Route 5
Route 6
Route 7
Route 8
Edit
Run

Event	Param
route_reservation	
route_freeing	
FRONT_MOVE_1	F
FRONT_MOVE_2	
BACK_MOVE_1	
BACK_MOVE_2	C
point_positionning	
route_formation	

Reserved blocks (resbl)
A
B
C
E
F
G
I

Reserved routes (resrt)
R5
R8

Reserved tracks (rsrtbl)	
Block	Route {R5,R8}
A	R5
B	R5
C	R5
E	R8
F	R8
G	R8
I	R8

Occupied blocks (OCC)
B
C

Lights

Green	Red
S5	S1
	S2
	S3
	S4

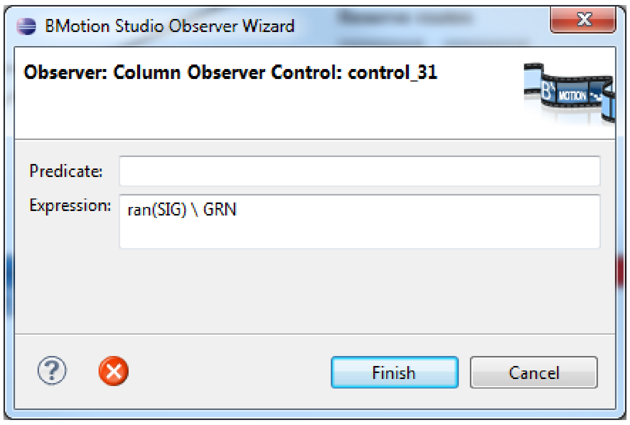
Name	Value
train_ctx0	
fst	{(R1↦A),(R2↦A)...
lst	{(R1↦C),(R2↦F)...
nxt	{(R1↦{(A↦B),(B...
rtbl	{(A↦R1),(A↦R2)...
train_ctx1	
SIG	{(A↦S1),(C↦S4)...
train_ctx2	
blpt	∅
lft	∅
rht	∅
train_0	
OCC	**{B,C}**
resbl	{A,B,C,E,F,G,I}
resrt	{R5,R8}
rsrtbl	{(A↦R5),(B↦R5)...
train_1	
LBT	{C}
TRK	{(B↦A),(C↦B),(...
frm	{R5,R8}
OCC	**{B,C}**
resbl	{A,B,C,E,F,G,I}
resrt	{R5,R8}
rsrtbl	{(A↦R5),(B↦R5)...
train_2	
rdy	{R8}
LBT	{C}
TRK	{(B↦A),(C↦B),(...
frm	{R5,R8}
OCC	**{B,C}**
resbl	{A,B,C,E,F,G,I}
resrt	{R5,R8}
rsrtbl	{(A↦R5),(B↦R5)...
train_3	
GRN	{S5}

invariant ok

no event errors detected

Figure 10.1: Application of the table control in BMotion Studio

For instance, Fig. 10.1 shows the application of the table control in a simulation of a train network using BMotion Studio. We see five different tables with different subsets of information about the current state in different fashions. For instance, the "'Reserved tracks (rsrtbl)" table gives the user an overview of the reserved blocks (first column) and the corresponding route of the reserved block (second column). Furthermore, the user sees at a glance the set of reserved routes in the top right cell. As already mentioned, the table control is fully customizable. The user is able to equip every cell, every column as well the entire table with information. For instance, the user can display a set or a relation in the table. This is done with simple observer and easy to use wizards in order to configure the observer. The user stays within a single notation, since BMotion Studio uses Event-B predicates and expressions as gluing code. For example, Fig. 10.2 shows the observer of the column "Red" of the "Lights" table. The observer defines the expression $ran(SIG) \setminus GRN$ where the result is the set of red lights. The observer arranges automatically the column of the table as shown in Fig. 10.1.


BMotion Studio Observer Wizard

Observer: Column Observer Control: control_31

Predicate:

Expression: ran(SIG) \ GRN

Figure 10.2: Column observer wizard

10.2 Other ProB Developments

When modelling cyber-physical systems it is often important to have access to a variety of mathematical functions, such as sine or cosine. These functions are not available "out-of-the-box" in either B or Event-B. However, ProB now supports externally defined functions, whose implementation can be provided in Prolog code. As such, one can provide an axiomatization of the required function for proving along with an executable version for simulation. This feature has, already been used in the Alstom interlocking case study of the Advance project. A screenshot of ProB simulating this case-study is shown in Figure 10.3. (It uses the Tk graphical visualization feature; a light-weight counter part of BMotion Studio for classical B which has also been improved during the course of the project).

In principle, this feature could also be used to call other simulators under the control of ProB.

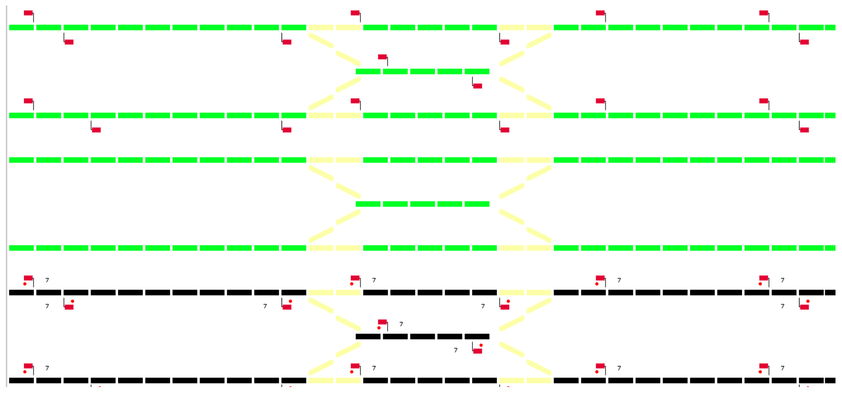


Figure 10.3: Animation of Alstom Interlocking Case Study

An important extension for Rodin is the Theory plug-in which allows the creation of extensions for the mathematical language. It supports new operators as well as datatypes. Using the theory plug-in, we can introduce concepts such as sequences or trees into the Event-B language. The theories are integrated into the proving infrastructure but not yet into other tools like ProB. We are currently working on an improved translation to support animation of user defined theories within ProB. The mathematical extensions are particularly important for modelling hybrid systems, as they enable Event-B models to access, both for proving and animation with ProB, to:

- real numbers or floating point numbers
- mathematical functions such as trigonometric functions
- they provide a way to access external functions in the ProB kernel, which could provide a link to other simulators.

Another important development is the support for recursive functions in ProB; these enable to provide animatable definitions for other kinds of mathematical theories.

The external and recursive functions were initially developed outside of the Advance project (direct funding by Alstom), but are now maintained, extended and integrated into Event-B as part of Advance.

Bibliography

[A$^+$05] Modelica Association et al. Modelica–a unified object-oriented language for physical systems modeling. *Language Specification, Version*, 2, 2005.

[ABHV06] J.R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An Open Extensible Tool Environment for Event-B. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006.

[Abr10] J. R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[ACB12] Edmunds A, J. Colley, and M. Butler. Building on the DEPLOY Legacy: Code Generation and Simulation. In *DS-Event-B-2012: Workshop on the experience of and advances in developing dependable systems in Event-B*, 2012.

[ASZ12] Jean-Raymond Abrial, Wen Su, and Huibiao Zhu. Formalizing hybrid systems with event-b. In John Derrick, John A. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *ABZ*, volume 7316 of *Lecture Notes in Computer Science*, pages 178–193. Springer, 2012.

[BCK12] J. Bendisposto, J. Clark, and P. Körner. ProB 2.0 Core API. Available at https://github.com/bendisposto/probcore, 2012.

[BLV$^+$10] J. F. Broenink, P. G. Larsen, M. Verhoef, C. Kleijn, D. S. Jovanović, K. G. Pierce, and F. Wouters. Design support and tooling for dependable embedded control systems. In *Proceedings of SERENE '10*, pages 77 – 82. ACM, ACM Sigsoft, April 2010.

[But09] M. Butler. Decomposition Structures for Event-B. In *Integrated Formal Methods iFM2009, Springer, LNCS 5423*, volume LNCS. Springer, February 2009.

[DLV12] P. Derler, E.A. Lee, and A.S. Vincentelli. Modeling Cyber-Physical Systems. *Proceedings of the IEEE*, 100(1):13 –28, jan. 2012.

[EB11] A. Edmunds and M. Butler. Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. In *PLACES 2011*, February 2011.

[Elb09] Nadine Elbeshausen. Highlevel Testing for ProB - Bachelor's thesis, Department of Computer Science, University of Düsseldorf, Germany, 2009.

[ERB12] A. Edmunds, A. Rezazadeh, and M. Butler. Formal Modelling for Ada Implementations: Tasking Event-B. In *Ada-Europe 2012: 17th International Conference on Reliable Software Technologies*, June 2012.

[Eve] Event-B State Machines. Details available at http://wiki.event-b.org/index.php/Event-B_Statemachines.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GLG$^+$11] R. Gmehlich, F. Loesch, K. Grau, J.C. Deprez, R. de Landtsheer, and C.Ponsard. DEPLOY Deliverable D38, D1.2 Report on Enhanced Deployment in the Automotive Sector. Technical report, Robert Bosch GmbH and CETIC, 2011.

[Jon90] C.B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1990.

[KGK$^+$07] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in Action.* Manning Publications Co., Greenwich, CT, USA, 2007.

[LB03] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *Proceedings of Formal Methods Europe 2003*, 2003.

[LBF+10] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The overture initiative integrating tools for vdm. *SIGSOFT Softw. Eng. Notes*, 35(1):1–6, January 2010.

[LBL09] Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. Visualising event-b models with b-motion studio. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Proceedings of FMICS 2009*, volume 5825 of *Lecture Notes in Computer Science*, pages 202–204. Springer, 2009.

[LLEL05] Xiaojun Liu, Jie Liu, Johan Eker, and Edward A. Lee. *Heterogeneous Modeling and Design of Control Systems*, pages 105–122. John Wiley & Sons, Inc., 2005.

[LSV98] E.A. Lee. and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12):1217 –1229, dec 1998.

[Mod] Modelica Association. Functional Mock-up Interface for Co-Simulation, Version 1.0. at https://svn.modelica.org/fmi/branches/public/specifications/FMI_for_CoSimulation_v1.0.pdf.

[Pla10] A. Platzer. Springer, 2010.

[RKW+10] A. Roth, V. Kozyura, W. Wei, A. Wieczorek, A. Fürst", T.S. Hoang, and J.Bryans. DEPLOY Deliverable D21, D4.1 Report on Pilot Deployment in Business Information Sector. Technical report, SAP AG, 2010.

[SAZ12] Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. Complementary methodologies for developing hybrid systems with event-b, 2012.

[SB10] Renato Silva and Michael Butler. Shared event composition/decomposition in event-b. In *FMCO Formal Methods for Components and Objects*, November 2010. Event Dates: 29 November - 1 December 2010.

[Sil11] R. Silva. Towards the Composition of Specifications in Event-B. In *B 2011*, June 2011.

[SPHB10] R. Silva, C. Pascal, T.S. Hoang, and M. Butler. Decomposition Tool for Event-B. *Software: Practice and Experience*, 2010.

[SS] C. Snook and V. Savicks. Event-B State Machines Animation. Available at http://wiki.event-b.org/UML-B_-_Statemachine_Animation.

[Thea] The Advance Project Team. The Advance Project. Available at http://www.advance-ict.eu.

[Theb] The DEPLOY Project Team. Project Website. at http://www.deploy-project.eu/.

[Thec] The Ptolemy Project. at http://ptolemy.eecs.berkeley.edu/.