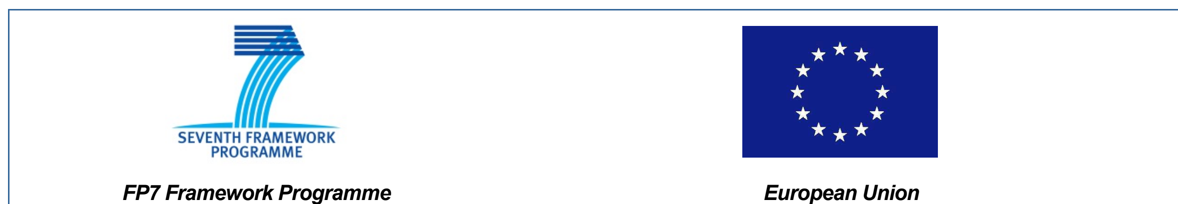
Advance

D.4.3 - METHODS AND TOOLS FOR SIMULATION AND TESTING II

ADVANCE

Grant Agreement: 287563
Date: 30/11/2014
Pages: 47
Status: Final
Authors: Michael Leuschel University of Düsseldorf
Reference: D4.3
Issue: 1

Partners / Clients:



FP7 Framework Programme | European Union

Consortium Members:

University of Southampton | Critical Software Technologies | Alstom Transport | Systerel | Heinrich Heine Universität | Selex ES

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

ADVANCE Deliverable D4.3

Methods and tools for simulation and testing II (revised)

Public Document

30.11.2014

`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

1 Introduction **5**

2 Theory-Plug-in Support **8**

2.1 Datatypes 8
2.2 Directly defined operators 9
2.3 Recursively defined operators 9
2.4 Axiomatic defined operators 10
2.5 Annotations for ProB 11
2.6 Theorems in Theories 14
2.7 Currently supported standard theories 14

3 Multi-Simulation Framework **16**

3.1 Co-simulation 16
3.2 Components Diagram Plug-in 22

4 Visualization **29**

4.1 Visualization of the State Space 29
4.2 Value over Time Visualization 32

5 Model-Based Testcase Generation **35**

5.1 Constraint-based test case generation 35
5.2 Evaluation of ProB performance on Setlog Test-case Generation Benchmarks 36
5.3 Minimum-Maximum Variable Coverage 41

6 Constraint Solving **43**

6.1 General Improvements 43
6.2 Random Enumeration 44
6.3 CHR 44
6.4 ProB as an SMT solver 44
6.5 Kodkod updates 45

Bibliography **46**

Chapter 1

Introduction

This deliverable describes the progress on the multi-simulation framework and the model-based testing infrastructure, in particular in response to demands by other workpackages and in response to decisions taken at the last review. One major decision was to use the FMI standard as the basis of our multi-simulation tool. This switch is one of the main contributions described in this deliverable. In the rest of this introduction chapter, we summarise this and other important contributions and developments achieved in the second period of the ADVANCE project. A more detailed account of the contributions can then be found in the later chapters.

Simulating Models with Theories

The Event-B language can be extended by using the Theory plug-in. Both industrial workpackages, WP1 and WP2 require mathematical extension to model the case studies. In WP1, the interlocking model requires the transitive closure operator for relations, e.g., to compute reachable track sections starting from a given train position. Similarly, in WP2 various mathematical extensions such as set summation are used to encode the smart grid case study. It is thus important that these mathematical extensions are supported by our multi-simulation tool in general, and ProB in particular. This has been successfully achieved in Chapter 2 of this deliverable.

Multi-Simulating According to the FMI-standard

In the first annual review it was suggested that we build the ADVANCE multi-simulation framework on the FMI standard. This undertaking was tackled in the second period of the review, and this work is described in Chapter 3 of the deliverable. A lot of work went into the ProB 2.0 Scripting

architecture, to enable one to coordinate multi-simulation of multiple units in Groovy or Java. Another notable development is the Components Diagram Editor, which enables one to compose FMUs within the Rodin IDE. With these contributions in place, the last period can now focus on using our multi-simulation framework for simulating cyber-physical systems in the smart grid and railway domain. High-level, discrete components can be described in Event-B and simulated with ProB, continuous or low-level hardware components can be simulated using FMI-compliant simulation tools.

Visualization

Visualization is important to quickly assess the behaviour or current state of a formal model. Various visualizations previously only available in the ProB Tcl/Tk version are now available for Event-B models within Rodin. Some new visualizations for multi-simulation were developed, in order to observe variable values over a certain trace of the system. This work is described in Chapter 4. A lot of effort went into the graphical, domain specific visualization using HTML and SVG technology. It allows, e.g., to render complex railway topologies for the formal models developed in WP1. This work is still ongoing and will be reported in the final deliverable.

Model-Based Testcase Generation

Various testcase generation features have been developed mainly according to the needs of WP5, involving an interplay between testing and safety analyses. The minimum-maximum coverage information arose out of the need to analyse the state explosion occuring during one WP5 case study, and has proven to be useful in providing a quick overview of the state space to the formal modeler. The existing constraint-based testcase generation algorithm has also been improved and made more generic. Due to prioritisation and the plan of work, there has been relatively little need for model-based testcase generation coming from WP1 and WP2 in the first two periods of the project. Still, we foresee more activities this area in the last period of ADVANCE, as model-based testing can provide substantial practical benefits of formal modelling.

Constraint Solving

The power and usefulness of many features of our multi-simulation and testcase generation framework depend on the performance of the underlying constraint solver. The effort that went into these tasks is summarised in

Chapter 6. Of interest here are the performance improvements for applications in the railway workpackage WP1, and the automatic detection of certain infinite comprehension sets, which greatly simplifies the modelling required by industrial users.

Summary

We have made substantial progress on our multi-simulation tool, which is now capable of seamlessly simulating cyber-physical system components. We have made major improvements in the scalability of ProB, both for simulation of high-level models, for complicated constraints in particular in light of model-based testing.

Chapter 2

Theory-Plug-in Support

The Theory plug-in [MBER10] provides the ability to append new data types and operators to Event-B's mathematical toolkit and extend Rodin's proof rules.

E.g. the railway sector case study [Col12] makes use of the Theory plug-in to define new mathematical operators and to facilitate the proofs.

We give a brief overview about how operators can be defined in a theory and how ProB can deal with that kind of definition. In the next paragraphs, we illustrate which ways the Theory plug-in offers to define new data types and operators and how ProB can handle them automatically. We also show the limitations of the approach and how ProB will be able to handle problematic operators by adding annotations manually.

2.1 Datatypes

The Theory plug-in allows to define recursive data types similar to algebraic data types known from functional programming languages. We take the theory of inductive lists as an example. A list is either empty or has a first element connected to the rest of the list which again is a list. We have two constructors:

- nil returns the empty list.
- $cons(head, tail)$ constructs the list which has $head$ as first element followed by the list $tail$.

Both constructors constitute a new operator in Event-B. The data type $List$ itself is also an operator which returns the set of all lists over a given set.

Lists makes also use of parametric polymorphism, another feature of theories. E.g. for the list example, we have a generic type parameter T to allow

lists for arbitrary sets. The argument *head* of the operator *cons* is of type T, the argument *tail* of type $List(T)$.

The datatypes of the Theory plug-in are similar to the free types of the Z notation. In a previous project [PL07] to support the Z notation by ProB we implemented an internal representation of free types. To re-use this effort we adapted the implementation in a way that it now also supports type parameters. For animation the constructors and destructors are directly replaced by the internal syntax constructs for free type constructors and destructors.

We can also encounter the situation that the type argument of a list is not only a type. E.g. the expression $List(1..9)$ specifies the set of lists whose elements can be integers between 1 and 9. Internally we use comprehension sets like

$$List(1 \mathinner{..} 9) = \{\ l \cdot l \in List(\mathbb{Z}) \wedge \\ (l = cons(h,t) \Rightarrow h \in 1 \mathinner{..} 9 \wedge t \in List(1 \mathinner{..} 9))\ \}$$

Note that this definition is recursive, in the comprehension set, we refer again to $List(1 \mathinner{..} 9)$. We explain the consequences of this in more detail below in section 2.3.

2.2 Directly defined operators

Directly defined operators can directly expressed by another predicate or expression. Let's take the theory of sequences as functions as an example. The set of sequences over a set S is defined by the operator $Seq(S)$. $Seq(S)$ is directly defined by the expression $\{n \mapsto f \mid n \in \mathbb{N} \wedge f \in 1 \mathinner{..} n \rightarrow S\}$. Thus, a sequence is a pair whose first element is the size of the sequence, and its second element is a total function which defines the elements at each position. Another example for a directly defined operator is the operator $seqIsEmpty(s)$ to check whether a sequence s is empty. It is defined by the predicate $\mathrm{prj}_1(s) = 0$.

We can animate the behaviour of such an operator by just replacing the operator with the given definition.

2.3 Recursively defined operators

Recursively defined operators are defined by a distinction of cases for an operator argument. E.g. the size $listSize(l)$ of an inductive list can be

defined recursively by:

$$\begin{array}{rcl} listSize(nil) & = & 0 \\ listSize(cons(head, tail)) & = & 1 + listSize(tail) \end{array}$$

We implemented recursive operators by translating an application of the operator to an B/Event-B function application with the particularity that the function is recursively defined:

$$\begin{array}{l} listSize(l) = \{\ a, r\ \cdot\ a = nil \Rightarrow r = 0\ \wedge \\ \qquad\qquad a = cons(head, tail) \Rightarrow r = 1 + listSize(tail)\ \}(l) \end{array}$$

Please note that we omitted existential quantifiers in the formula above to make it more readable. If an operator defines a predicate instead of an expression, we replace the application of the operator by a membership test.

ProB's support for recursive functions was limited to global functions that depend not on a current state. But the theory plug-in's datatypes can be used in a more general style. E.g. the expression $List(X)$ specifies a the set of lists whose elements in the set X. But X can be a variable of the machine or even a parameter of an operation. Thus we needed a more flexible way to specify recursive functions. We introduced a new element $recursive(I, S)$ in the internal abstract syntax tree of ProB that allows us to define an identifier I that refers to the specified comprehension set S. We had to adapt the internal datastructure that represents symbolic sets such that the recursive definition is respected when evaluating the set.

The translation of $e \in List(X)$ with X beeing a set of integers is now.

$$\begin{array}{l} e \in \text{recursive}(L, \{\ l\ \cdot\ l \in List(\mathbb{Z})\ \wedge \\ \qquad\qquad (l = cons(h, t) \Rightarrow h \in X \wedge t \in L)\ \}). \end{array}$$

2.4 Axiomatic defined operators

The most flexible approach to define the behaviour of an operator is to specify a set of axioms. We currently do not see any feasible way to provide generic support for expressions that use these operators.

An example for an axiomatic definitions is the summation operator, whose behaviour is defined by the following three axioms:

$$\begin{array}{ll} \text{axm1} & SUM(\varnothing) = 0 \\ \text{axm2} & \forall t, x \cdot t \in T \wedge x \in \mathbb{Z} \Rightarrow SUM(\{t \mapsto x\}) = x \\ \text{axm3} & \forall s, t \cdot s \in T \mathrel{+\!\!\!\rightarrow} \mathbb{Z} \wedge t \in T \mathrel{+\!\!\!\rightarrow} \mathbb{Z} \wedge s \cap t = \varnothing \\ & \qquad \Rightarrow SUM(s \cup t) = SUM(s) + SUM(t) \end{array}$$

2.5 Annotations for ProB

In the previous sections, we explained how operators can be animated by analysing their definition. For practical purposes it can be more effective to instruct ProB directly how an operator should be handled. We give two examples where an alternative to the standard behaviour described above is preferred.

2.5.1 Transitive closure

The *closure* operator which returns the transitive closure of a relation r is defined by using a direct definition:

$$closure(r) = fix(\lambda s \cdot s \in S \leftrightarrow S \mid r \cup (s;r))$$

The fixpoint operator fix is also defined by a direct definition

$$fix(f) = inter(\{s | f(s) \subseteq s\}).$$

In summary, the operator could theoretically be handled automatically by ProB by replacing $closure(r)$ with the expression

$$inter(\{s | r \cup (s;r) \subseteq s\}).$$

In practice, the number of possibilities for the quantified variable s in the expression becomes large very fast. If s is of type $T \leftrightarrow T$, s has $2^{|T|^2}$ possible values. For $|T| = 4$, ProB must check 65536 sets, for $|T| = 5$ already more than 33 million sets. Thus for most models, ProB is not capable of handling this direct definition effectively.

On the other hand, ProB has already built-in support for classical B's `closure1` operator. To handle the closure operator effectively, we have added an annotation to the operator definition that instructs ProB to use its built-in closure support (rather than the direct definition). The built-in closure support has no problem dealing with large relations, as the following transcript from ProB's REPL (Read-Eval-Print-Loop) shows:

```
>>> f=closure1(%x.(x:1..5000|x*x)) & f[{2}] = r
Existentially Quantified Predicate over f,r is TRUE
Solution:
       f = #5077:{(1|->1),(2|->4),...,(4999|->24990001),(5000|->25000000)} &
       r = {4,16,256,65536}

>>> g=closure1(%x.(x:1..5000000|x*x)) & g[{2}] = r
Existentially Quantified Predicate over f,r is TRUE
```

```
Solution:
       g = closure1(%x.(x : (1 .. 5000000)|x * x)) &
       r = {4,16,256,65536,4294967296}

>>> h=closure1(%x.(x:NATURAL|x/2)) & h[{2**40}] = r
Existentially Quantified Predicate over h,r is TRUE
Solution:
       h = closure1(%x.(x : NATURAL|x / 2)) &
       r = {0,1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,
       65536,131072,262144,524288,1048576,2097152,4194304,8388608,16777216,
       33554432,67108864,134217728,268435456,536870912,1073741824,
       2147483648,4294967296,8589934592,17179869184,34359738368,
       68719476736,137438953472,274877906944,549755813888}
```

The first expression shows that the transitive closure `f` of a 5000 element relation can be computed quickly (in about 50 ms). The last two expressions show that, for large or infinite relations, ProB reverts to computing the closure lazily on demand. The computation is instantaneous (10 ms or less).

2.5.2 Sum and Product

The sum operator as described above cannot be animated by ProB without additional information because axiomatic definitions are not supported. By explicitly instructing ProB we can compute the sum operator by using the classical B sum operator Σ:

$$SUM(s) = (\Sigma t, x \cdot t \mapsto x \in s | x)$$

Currently, ProB supports a theory with the sum operator together with a product operator (Fig. 2.1). We have added an annotation that instructs ProB to use its built-in support for sum and product.

2.5.3 Implementation of the Annotation Mechanism

ProB checks if there exists a file with the name *≪theory_name≫.ptm* in the same directory of the theory and reads its content. A first version only allows tags that show ProB that this is an operator with an alternative implementation. E.g. "SUM is the summation operator".

We examine upcoming theories to check whether a more flexible approach is beneficial. E.g. the file could contain instructions how the value of an operator can be computed effectively. We currently do not see a need to provide this feature for the theories known to us.

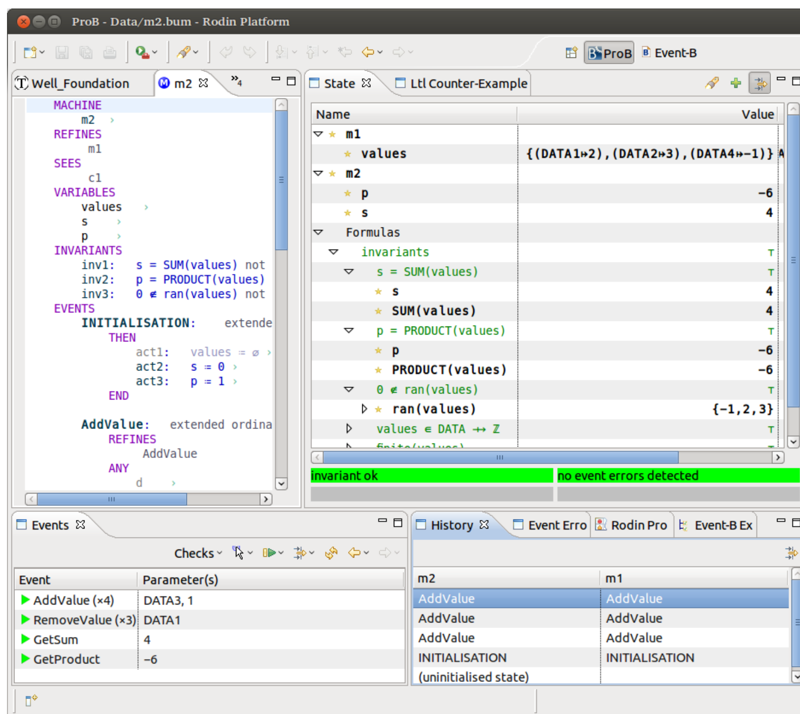


Figure 2.1: Screenshot of an animation using the theory of sum and product

2.6 Theorems in Theories

A theory might also contain arbitrary theorems. They are usually provided to facilitate proofs. ProB completely ignores the theorems because it is currently not our goal to check the correctness of the theory but to animate the models that use it.

In future, it could be interesting to provide support for checking theories, too. This would be especially helpful for users who define their own theories.

2.7 Currently supported standard theories

The developers of the Theory plug-in contributed a project with a set of theories. These are candidates for standard theories when the next version of the plug-in will be released.

All operators defined in these standard theories **are now supported** by ProB:

- "Sum and Product" defines operators to compute the sum or product of integer sets. It is fully supported by specific tagged operators as explained above (2.5.2).
- "Binary Tree" defines a new polymorphic data type to represent binary trees and operators on these. The structure is very similar to the "List" theory. It uses recursively defined operators, all operators are supported.
- "Bool Ops" defines operators on Boolean values (AND, OR, NOT). It uses direct operator definitions, all operators are supported.
- "Fix Point" uses a direct operator definition, is theoretically supported but usually to complex to animate.
- "List" defines a polymorphic datatype and operators on lists. The operators are all recursively defined, all operators are supported.
- "Main" contains no operators, just theorems and proof rules which is not relevant for animation.
- "Seq" is a theory over sequences. All operators are defined by direct definitions and are supported (see 2.2).
- "closure" is a theory which defines an operator that yields the transitive closure of a relation. It supported by a specific tagged operator (see 2.5.1).

There is some demand within the Advance project coming from WP2 to support a theory of real numbers. This would allow to express certain models in Event-B rather than requiring co-simulation with continuous models. This would, however, require considerable implementation effort to extend the ProB kernel for real numbers and the associated operators (and is almost a research project on its own).

Chapter 3

Multi-Simulation Framework

3.1 Co-simulation

As mentioned in D4.2 [?], in order to enable co-simulation of discrete models in Event-B with continuous models of the environment, developed in other languages and tools, we have decided to use an industrial level tool-independent standard *Functional Mock-up Interface*[1], which specifies cross-platform API for the exchange and simulation of dynamic models. The *Co-Simulation* part of the standard is based on the concept of a master-slave architecture, where the master is responsible for data exchange and synchronisation of the simulation of all slaves, i.e. subsystems exported from different environments as Functional Mock-up Units (FMUs). In order to reflect this architecture we have defined our master algorithm and slave semantics in a formal way.

Our generic master has been designed based on an algorithm example from the FMI specification document [?]. The abstract form of the algorithm within the context of discrete-continuous co-simulation is shown in Figure 3.1[2]. Simulation process is divided into fixed-size steps with the discrete communication points at their boundaries. *CoSim* master is responsible for synchronising slaves and communicating the data (*ReadWrite*), as well as for calling simulation steps. We distinguish the continuous step (*CStep*) of FMUs and the discrete step (*DStep*) of Event-B models, with the latter realised by executing a sequence of events that model a single simulation cycle, followed by a blocking *Wait* event to synchronise the next step.

For synchronisation of the slaves the master keeps a record of the global simulation time. The state of an individual slave with respect to time can

[1] `https://www.fmi-standard.org`

[2] Read the graph from top to bottom, left to right

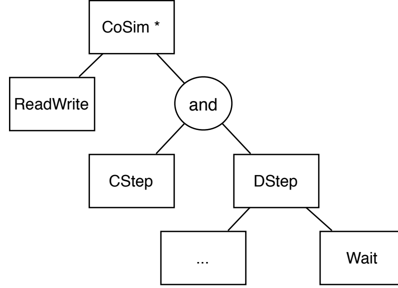


Figure 3.1: Abstract view of the co-simulation master

therefore be defined as a function:

$$F\colon Time \rightarrow V \tag{3.1}$$

where V is the state of the slave's internal variables. The evolution of each variable and its slave over time can be represented on a graph:

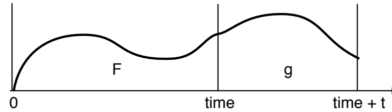


Figure 3.2: The state of a slave over time

where g is a state function defined over time interval $time \ldots time + t$. If t equals the communication step of the master, the simulation semantics of a slave can be formally defined using Event-B notation as follows:

machine C
variables $F, time$
event $CStep =$
any i, t, g
where
$g \in [time \ldots time + t] \rightarrow V$
$g(time) = F(time)$
$P(g, i, F, time, t)$
then
$time := time + t$
$F := F \cup g$

where parameter i is the slave's inputs and P is the model properties, or predicates that g must satisfy. This formal model specifies the semantics of continuous slaves, as it depends on time and continuous function F. For the

discrete slaves of Event-B we can derive a simpler formalism that depends purely on input and internal variables:

machine D
var V, O
event $DStep =$
any i
where
$i \in T$
then
$V, O := S(V, O, i)$

where i is the in[put], O is internal ... is a discrete state f... g[lobal] ... synchronise them w... [sl]aves th... must be an existing Event-B machine ... the end of a simulation cycle.

The above conce[pt is illus]trated ... consists of a plant (w...) with a c... that controls the valve of input flow to maintain the desired water level:

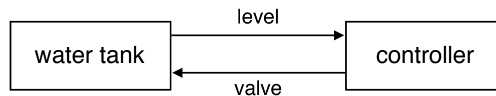


Figure 3.3: Controlled water tank model

The plant and controller can be modelled as a continuous and discrete slave, accordingly, which exchange a continuous signal *level* and a discrete signal *valve*. Given the formalism above, the control flow of the simulation can be represented by a refined graph in 3.4.

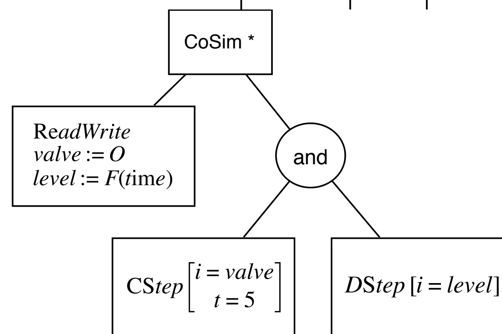


Figure 3.4: Control flow of the discrete-continuous co-simulation

The diagram does not display the Wait event, which is part of the discrete (Event-B) component. However, it is important to note that Wait must be the last enabled event within the simulation step. It may be either a single event or multiple events, in which case only one of them will be executed. Additionally, at the very abstract level a single Wait event can represent the whole simulation step and later be refined to multiple events.

In a more concrete form our master algorithm is designed to comply with FMI standard (though it is not part of the standard itself), i.e. we have developed it to reflect the use of FMI API. The outline of the master algorithm is as follows:

1. Instantiate all slaves: `Component.instantiate())`
2. Initialise all slaves: `Component.initialise(startTime, stopTime)`
3. Set global time to a start time and begin the simulation loop
4. Read all outputs: `Component.writeOutputs())`
5. Write all inputs: `Component.readInputs())`
6. Perform simulation step: `Component.doStep(time, stepSize))`
7. Increase time by step size; if time has reached the stop time then stop, otherwise go back to step 4
8. Terminate slaves: `Component.terminate())`

The FMI API defines a method for each action on a slave, thus master only needs to operate with API. We followed the same approach by defining a similar API for Event-B slaves and moved it to an abstract slave, which we call *Component*. Thus both Event-B and FMI components implement the same API, which greatly simplifies the master and allows us to extend co-simulation to other types of components without modifying the master. The metamodel that reflects the data structure of co-simulation framework and relationships between involved entities is shown in Figure 3.5.

As you can see the simulation API is defined in the abstract class Component, and consists of key methods, required by the master: `instantiate`, `initialise`, `writeOutputs`, `readInputs`, `doStep` and `terminate`.

For FMU components the mapping to implementation of our API is straightforward, as it directly maps to corresponding FMI API. In order to operate FMI models from Rodin we have implemented an interface in the ProB Core using the JFMI Java wrapper[3] on top of the FMU SDK[4].

[3]`http://ptolemy.eecs.berkeley.edu/java/jfmi/`

[4]`http://www.qtronic.de/en/fmusdk.html`

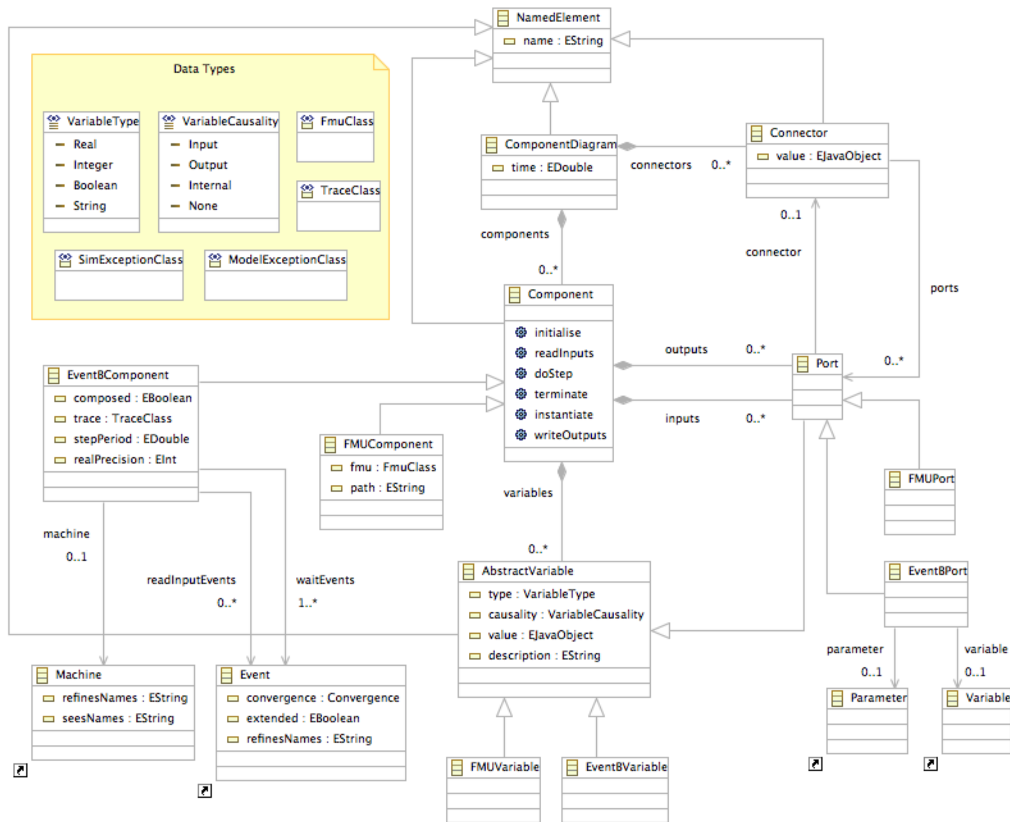


Figure 3.5: Metamodel of the co-simulation framework

For the implementation of Event-B components we used the formalism described earlier, which maps the simulation step marked by Wait event(s) to the method `doStep`, essentially executing a sequence of events non-deterministically until one of the Wait events is enabled (multiple events can be defined as Wait events). The data exchange is implemented by `writeOutputs` and `readInputs` methods. The former is implemented by executing one of the special *ReadInput* events of Event-B machine (again, multiple events can be defined as ReadInput events, but they must match in the number and name of parameters). The `writeOutputs` method is implemented by simply reading a value of the corresponding Event-B variable to the output port, as reading variables does not require an event.

To demonstrate how Event-B machines map to Event-B Components used in the simulation we show a simple refinement of a water tank valve controller model in Event-B:

machine $tankController0$
variables $valve$
events

$$\left.\begin{array}{ll} SwitchOn = & \textbf{any } l \textbf{ where } l < LT \textbf{ then } valve := on \textbf{ end} \\ NoSwitch = & \textbf{any } l \textbf{ where } l \geq LT \wedge l \leq HT \textbf{ then } skip \textbf{ end} \\ SwitchOff = & \textbf{any } l \textbf{ where } l > HT \textbf{ then } valve := off \textbf{ end} \end{array}\right] DStep$$

end

In this abstract machine all three events are ReadInput events and Wait events. Only one of these events gets executed in a simulation cycle, depending on the value of l that is an input signal from the plant.

When $tankController0$ is refined to introduce multiple sequential discrete steps, then the ReadInput and Wait events become distinct. $tankController0$ could be refined as a state machine as follows:

machine $tankController1$ **refines** $tankController0$
variables $valve, level, state$
events

$$\left.\begin{array}{l} ReadLevel = \textbf{ any } l \textbf{ where } state = 0 \textbf{ then } level := l \textbf{ end} \\ \\ DecideOn = \textbf{ where } state = 1 \wedge level < LT \textbf{ then } state := 2 \textbf{ end} \\ DecideSkip = \textbf{ where } state = 1 \wedge level \geq LT \wedge level \leq HT \\ \quad \textbf{then } state := 3 \textbf{ end} \\ DecideOff = \textbf{ where } state = 1 \wedge level > HT \textbf{ then } state := 4 \textbf{ end} \\ \\ \left.\begin{array}{l} SwitchOn \textbf{ refines } SwitchOn = \\ \quad \textbf{witness } l = level \textbf{ where } state = 2 \textbf{ then } valve := on \textbf{ end} \\ NoSwitch \textbf{ refines } NoSwitch = \\ \quad \textbf{witness } l = level \textbf{ where } state = 3 \textbf{ then } skip \textbf{ end} \\ SwitchOff \textbf{ refines } SwitchOff = \\ \quad \textbf{witness } l = level \textbf{ where } state = 4 \textbf{ then } valve := off \textbf{ end} \end{array}\right] Wait \end{array}\right] DStep$$

end

In the refinement *ReadLevel* is a ReadInput event while $SwitchOn$, $NoSwitch$ and *SwitchOff* are only Wait events. This flexibility of indicating multiple and/or same events as ReadInput and Wait events enables the refinement of control events and, most importantly, verification and co-simulation of Event-B components from the early stage of development, which is crucial for safety-critical systems.

3.2 Components Diagram Plug-in

The key elements required for the FMI co-simulation are the master algorithm, described in 3.1, and a component-connection graph that captures the topology of subsystem (slave) composition into a complete simulated model. The graph is used by the master to perform data exchange and coordination of the simulation of all components. For our co-simulation framework we have designed a visual diagram editor and simulation surface for Rodin that allows to import components (currently Event-B or FMU) to the diagram as nodes, connect them via input/output ports, validate the resulting composition and control/display the simulation state with time. Below we demonstrate the capabilities of the tool and the results of co-simulation experiments with one of the leading modelling and simulation environments Dymola[5].

In order to create a composed model for co-simulation we first need to create an empty components diagram and import the required components. A diagram can be created by going to the standard new elements wizard: `File->New->Example->Components Diagram`. The diagram always consists of two files: the diagram file itself `.cmd` and a corresponding domain file `.cmp`, both of which will be created in a workspace folder of our choice. Note that the Event-B Explorer view in Rodin does not show any resources apart from Event-B. To see the diagram files either resource view filters must be turned off or another project view can be used, such as the Project Explorer or Navigator. When the diagram is created it is opened in the components editor automatically (Figure 3.6).

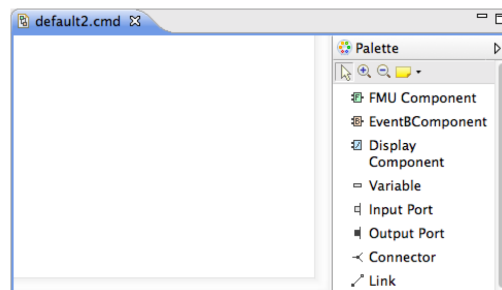


Figure 3.6: An empty components diagram

To import a component we use the Import button on the diagram toolbar (in the main Rodin toolbar). The button opens a Component Import Wizard (Figure 3.7), where we can either browse the file system or the workspace for

[5]`http://www.3ds.com/products-services/catia/portfolio/dymola`

components. Currently only Event-B machines and FMUs are supported for import.

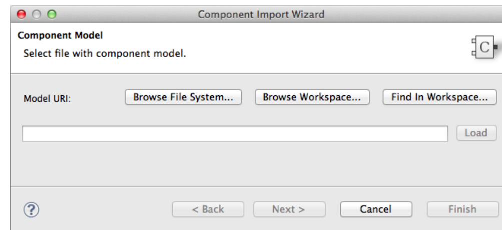


Figure 3.7: Component import wizard

Depending on the selected component (Event-B or FMU) a corresponding component definition wizard page is loaded. If we select an FMU file and press Next, the FMU Component Definition page is displayed (Figure 3.8), which shows all the variables declared in the loaded FMU, grouped into inputs, outputs and internal variables. The variables of interest that we want to observe during the simulation can be selected here, before the Finish button is pressed and a component node is created on the diagram.

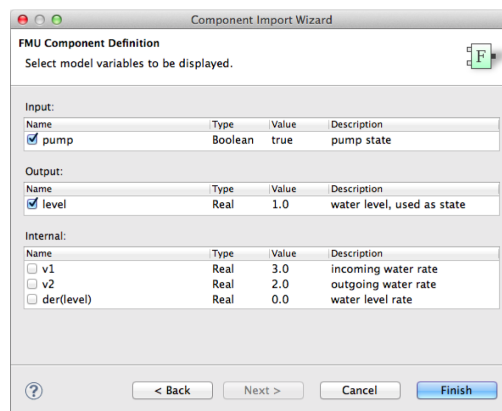


Figure 3.8: FMU component definition page

In case an Event-B machine file is selected, first, the Event-B Parameter Definition page is loaded (Figure 3.9), which lets us to define the simulation step period of the Event-B component, Real-type signal conversion precision (magnitude of the conversion from FMU Real type to Event-B integer and back), a dedicated Event-B variable for time (optional), and ReadInput and Wait events. The wizard cannot be progressed until the Wait events are added. The ReadInput events are optional, as some components may not have any input ports.

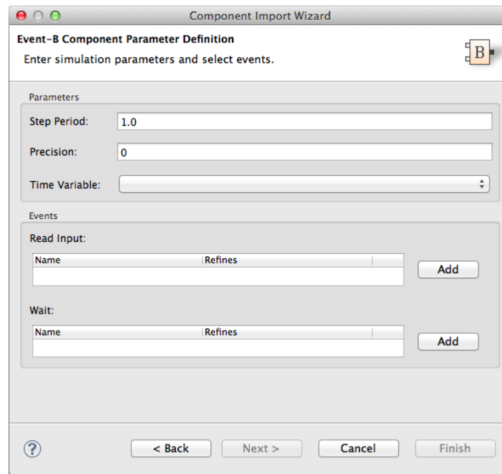


Figure 3.9: Event-B component parameter definition page

The second definition page of Event-B components is the Variable Definition page (Figure 3.10), which lets us to add input/output ports to a component and select Event-B variables to be displayed on the diagram. When adding ports a pop-up window is shown where we have to specify correct port type (Real, Integer, Boolean or String) and either a ReadInput event parameter (for input ports) or Event-B variable (for output ports).

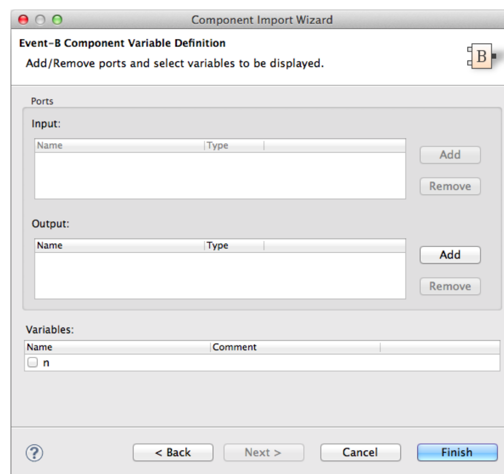


Figure 3.10: Event-B component variable definition page

When the configuration of Event-B component is finished and the component is imported, saving the diagram will also save our configuration in the corresponding Event-B machine, which will be used in succeeding imports of

the same component (it can be still reconfigured if necessary).

After components are imported they can be connected via input/output ports using Connectors. Connectors are required due to the limitations of the GMF framework that was used to implement the diagram. To connect an output port to an input port a Connector element needs to be selected from the diagram palette and added to the diagram. A Link element then can be used to connect a port to a connector. Only a single link is allowed per port and a single output can be linked to a connector. The attributes of any component can be modified at any point in the Properties view.

When done with the connection graph the system can be simulated either in a single run (Simulate button) or step-by-step (Simulate Step button). At the beginning of the simulation a dialog is displayed, where we can enter the duration of the simulation and the step size (Figure 3.11).

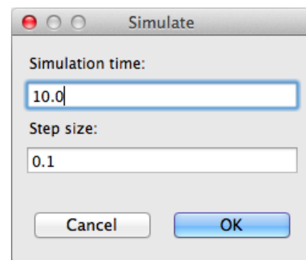


Figure 3.11: Simulation dialog

If the Simulate Step mode is selected the simulation will be executed for one step and paused, at which point the next step can be simulated by repeated press of the Simulate Step button, or the system can be simulated until the end time by pressing Simulate. The values of all the variables, ports and connectors are updated between the simulation steps and displayed on the diagram. Complete results of the simulation, i.e. the time of communication points of the master and values of all the variables, are written at the end of simulation into a `results.csv` file in the same directory where the diagram is located. Additionally, signals can be plotted over time using the Display component, which can have only input ports. When double-clicked at any time before, during or after simulation it shows a plot of the connected input signals. During the simulation the plot is updated in real time.

To test the functionality of our framework we have conducted a number of experiments with the available simulation environments that support the FMI standard. One of the best tools in our opinion is a Modelica-based commercial environment Dymola. We took the same water tank example mentioned previously and modelled it in the Dymola using only components

of the Modelica Standard Library and its sub-libraries Fluid and StateGraph. The model consists of the plant and controller parts. The plant was modelled by a tank, a water source boundary under pressure, connected to the tank via discrete valve, and an ambient pressure water sink boundary. The controller was modelled using the StateGraph library as a two-state state machine, with states valveOn and valveOff and input signal triggered transitions, linked to level threshold predicates. The complete model is displayed in Figure 3.12.

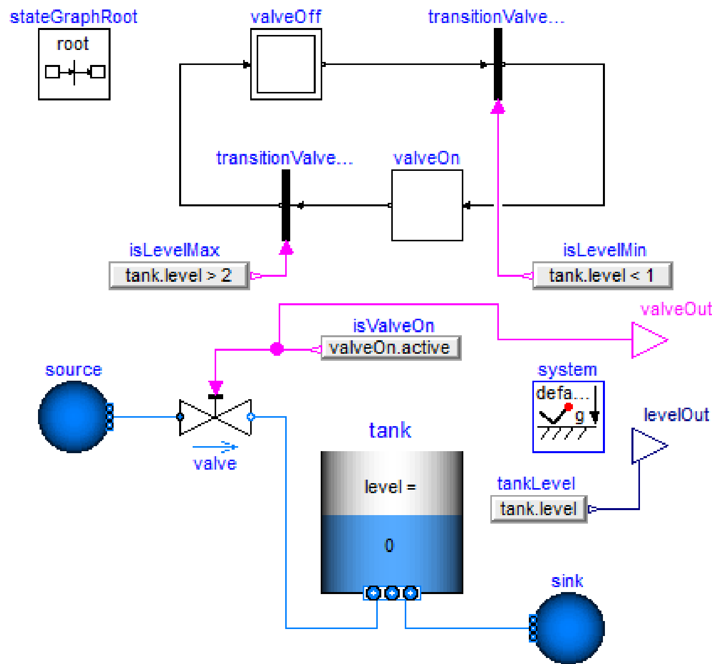


Figure 3.12: Controlled water tank model in Dymola

The model was initialised using the parameters for the water tank and port dimensions, pressure and temperature in the source and sink, and valve pressure loss. The medium of all fluid elements was set to a ConstantPropertyLiquidWater. Tank water level thresholds were set to 1m and 2m. The model was validated and simulated, yielding the results plotted in Figure 3.13.

The next step was to split the model into a plant and controller, and, as we were only interested in the continuous model of the plant, export the plant subsystem as an FMU for Co-simulation. The model of the plant is shown in Figure 3.14. The only difference from the complete model is the replacement of the valve input signal expression with the external input signal.

For the controller subsystem we took Event-B models described in 3.1 and imported them as Event-B components, specifying ReadInput and Wait

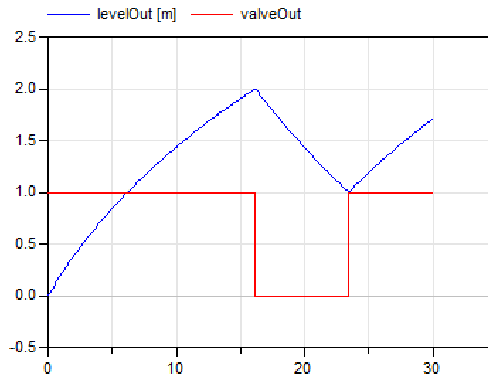


Figure 3.13: Co-simulation results of the controlled water tank model in Dymola (simulation time = 30s, step size = 0.1s)

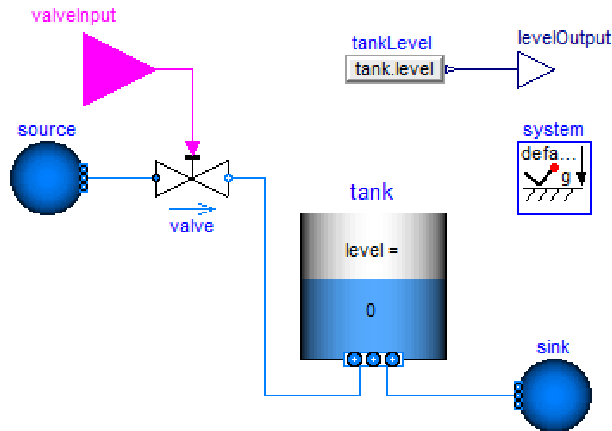


Figure 3.14: Controlled water tank plant subsystem

events and adding a Real-type input port (linked to a parameter l of the ReadInput event *readLevel*) and a Boolean-type output port (linked to variable *valve*). The plant FMU was also imported and configured by unchecking all internal variables and leaving only input port *valveInput* and output port *levelOutput*. To visualise the signals a Display component was added. The screenshot of the final composition diagram is shown in Figure 3.15.

The diagram was simulated for the same duration and step size as in the Dymola. The simulation results that were plotted via the Display component are shown in Figure 3.16. Comparison with the results obtained from the Dymola environment (Figure 3.13) demonstrates that the dynamics is simulated as expected, although there is a slight miss of the water threshold due to a relatively big communication step size. In fact, this indicates a problem

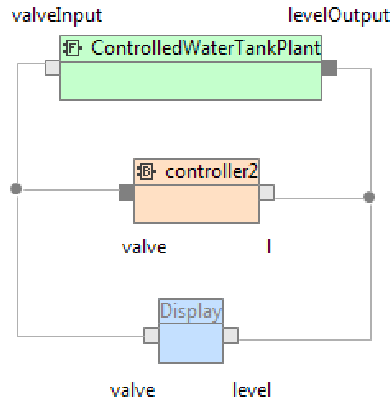


Figure 3.15: Component diagram of the controlled water tank

with our original Dymola model of controller, as it did not take into account both sensor and controller delays. In case of a discrete Event-B controller with a fixed step period that omission translates directly to a delay in the control signal. The issue can be either minimised by reducing the step size, or by modelling a predictive controller that takes into consideration all possible communication delays, which is a more realistic solution. This problem proves the importance of the validation of Event-B models against a precise model of environment, and our co-simulation plug-in addresses exactly that.

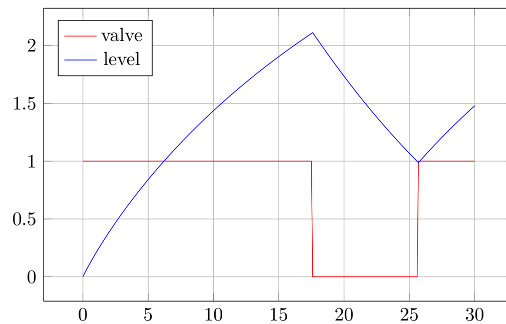


Figure 3.16: Co-simulation results of the controlled water tank model in the multi-simulation framework (simulation time = 30s, step size = 0.1s)

Chapter 4

Visualization

The effectiveness of new visualization techniques developed to aid understanding of large-scale simulations have been evaluated and measured. One of the main areas that we have focused on is the generation of simple visualizations based on the data that is produced by ProB during animation. These visualizations will allow the users both to better understand the behavior of their specifications as well as to explain that behavior to others. The algorithms that we used were already available in ProB, but we have now integrated them into the ProB Rodin plugin. Furthermore, previously, the visualizations were statically generated. Now they are dynamically updated when changes in the current animation take place.

4.1 Visualization of the State Space

One of the main visualizations that we focused on was the dynamic creation of a visualization for the state space of the current animation. This visualization uses a force based layout to visualize the states in the state space and the operations between given states. The visualization does not require the complete exploration of the state space before its creation. Instead, the visualization of the state space is created using the known transitions and states and then updated when new states are discovered.

Status about the invariant is available in the visualization based on the color of the nodes. If an invariant violation is present, the state is colored red. If the invariant is ok, the state is colored green. Otherwise, if the invariant has not yet been calculated for the given state, the node is colored gray. The labels for the states are the values of the variables for the given state. An example of this visualization can be seen in Figure 4.1.

4.1.1 Reduced State Spaces

Because of the state space explosion problem, a visualization of the entire state space can be too much information for a user to process at any given time. In order to help understand the overall behavior of the state space, we have also made smaller graphs available that are derived from the original state space and can be more useful for the user. Both of the algorithms that we used were adapted from the visualizations that are currently available in the Tcl/Tk version of ProB.

Signature Merged State Spaces

The signature merge algorithm is shown in Figure 4.2. The algorithm works by merging all of the states which have the same outgoing transitions. This creates a state space that is considerably smaller but that still preserves information about the operations that are enabled for a given state. By default, the algorithm considers all of the transitions that are present in the model. However, it is also possible for the user to specify a given subset of transitions that are of interest.

Transition Diagrams

The other reduction algorithm that is supported is the creation of transition diagrams. In order to perform this algorithm, ProB receives an expression from the user and calculates all of the possible solutions to the expression in the scope of the model that is being animated. These become the vertices in the graph. The edges in the graph show the transitions that change the value of the given expression to that of the value shown in the target state (see Figure 4.3).

4.1.2 User Customization

We also wanted the user to be able to customize the appearance of the visualizations. This is particularly the case with the state space visualization because there is so much information that has to be visualized. In order to provide the user with this functionality, we have integrated the visualization framework into the existing Groovy console in order to enable the user to customize their visualization. The customization works by allowing the user to define the states that they wish to transform and the transformations that they wish to apply. For instance, the user can select states that match a specified predicate and change their color.

Figure 4.1: Section of the state space visualization for a scheduler example

Figure 4.2: Reduced state space using the signature merge algorithm

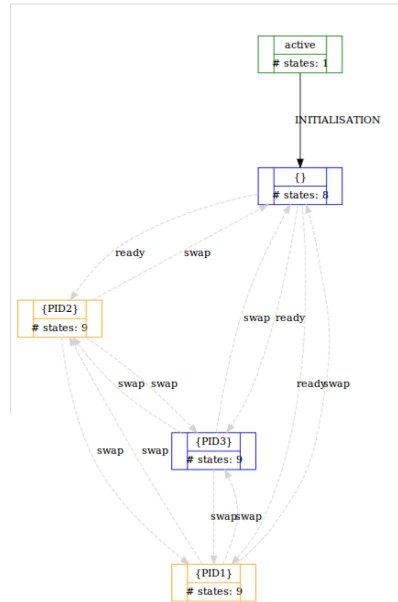


Figure 4.3: Transition diagram for a scheduler example

4.2 Value over Time Visualization

Because of the focus on the multi-simulation of specifications over a given time period, we were interested in creating a visualization that would be able to provide the user with information concerning the behaviors of the animation over that given time period. In order to do this, we created a simple line plot which plots the value of a user specified formula over a given animation. By default, the values of the formula are plotted against the number of animation steps in the given trace, but it is also possible for the user to specify an expression that represents time for the given specification. In this case, the values of the formula over the given trace will be plotted against the value of the given expression.

There are two modes for viewing the line plots. The first mode plots all of values in the same line plot which allows the user to visualize how the values of the formulas react in relation to each other. The second mode creates a separate line plot for each specified formula. This is useful for a user who wants to visualize several formulas that may not necessarily be related. It is possible to visualize predicates and expressions that take on integer or boolean values. If the formula that is being visualized is a predicate or an expression that takes on a boolean value, the resulting value is mapped to

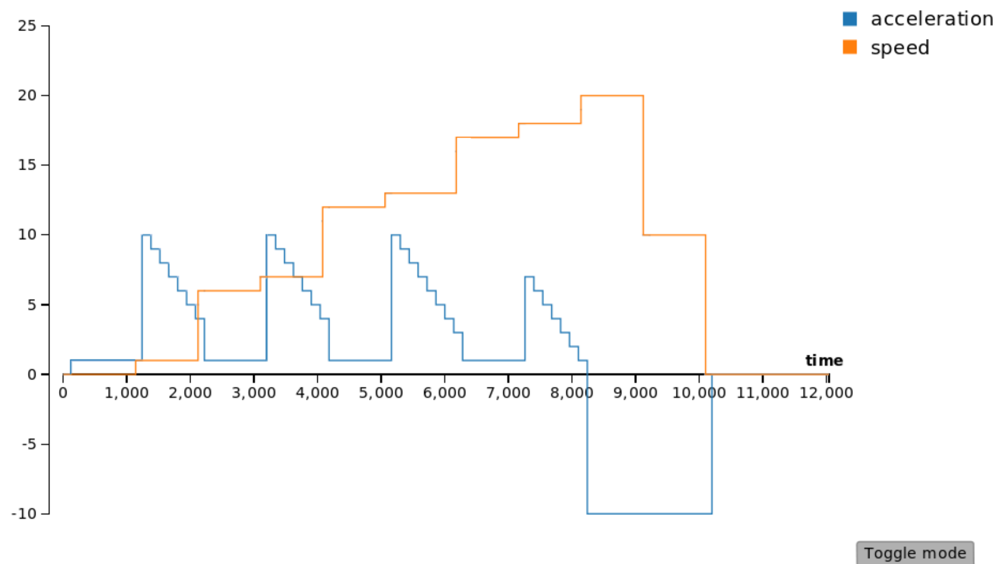


Figure 4.4: Value over time visualization

an integer value. If all of the formulas are being drawn in the same line plot, the boolean value TRUE will be mapped to the maximum value of all the data sets. This allows the user to see if the change in a predicate or boolean expression resulted in the change of the formula with an integer value. If the boolean formula is plotted separated, the values TRUE and FALSE are mapped to 0 and 1 respectively. An example of this visualization can be seen in Figure 4.4.

We also created a visualization for the closer inspection of a given formula. ProB already includes support for the generation of a DOT file which, when rendered, breaks the displays a formula as a tree in which the direct subformulas of a given formula are children nodes [LSBL08]. The formulas and its subformulas are also evaluated for the current state in the animation, and the resulting representation of the formula is colored so as to specify its value (e.g. if a predicate evaluates to true at the specified state, the predicate is colored green). This enables the user to identify the parts of a formula that may be problematic. We recreated this visualization within ProB 2.0. The new visualization is also interactive. The user can click on a node to expand or contract that part of the tree. This allows the user to really focus on the parts of the formula that are of interest.

Both of the visualizations that are described above are based on the current state in a given animation. When an animation step in an animation

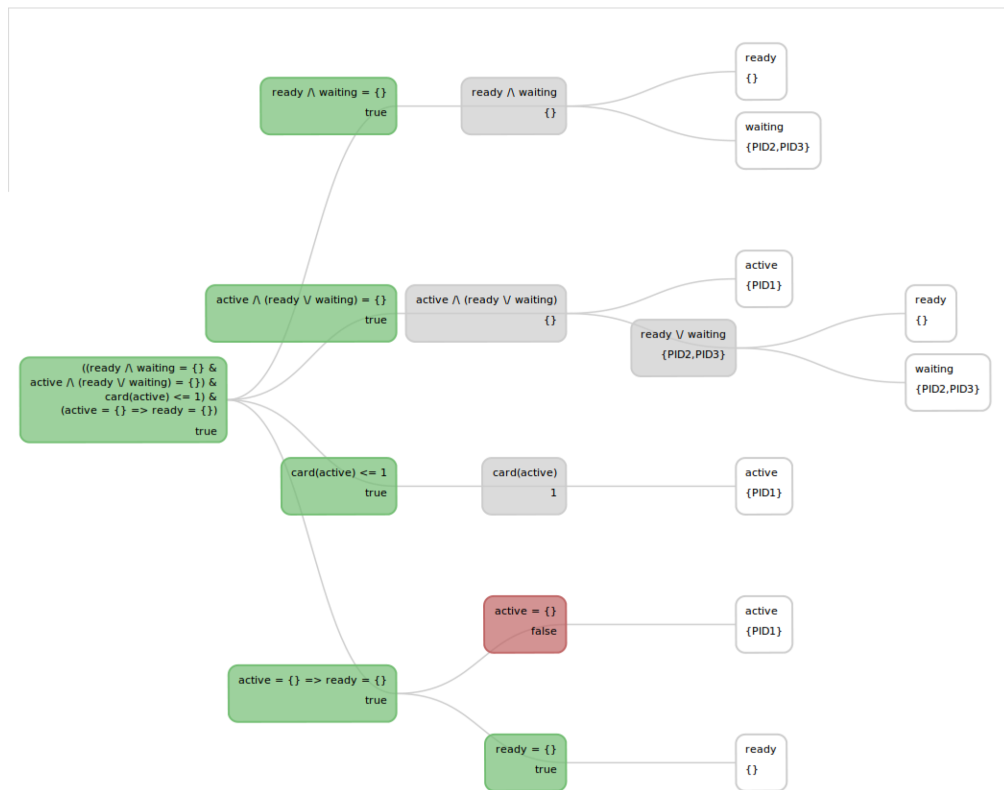


Figure 4.5: Visualization of the invariant for a scheduler example

occurs, they are automatically updated to display the data for the current state in the animation. An example of this visualization can be seen in Figure 4.5.

Chapter 5

Model-Based Testcase Generation

Various testcase generation features have been developed in this period, mainly according to the needs of WP5, involving an interplay between testing and safety analyses.

5.1 Constraint-based test case generation

Figure 5.2 contains the ProB constraint-based test-case generation algorithm, which is at the basis of the developments within the Advance project. The algorithm uses the constraint solver to find longer and longer event sequences, satisfying event coverage with additional target predicates.

Thus far we have already achieved the following goals:

- the algorithm can now be called from the ProB command-line tool using the command `-cbc_tests Depth EndPredicate File`. The coverage criterion can be specified using the commands `-cbc_cover Event` or `-cbc_cover_all` if all events should be covered.
- the Prolog code corresponding to Figure 5.2 has been refactored for allowing the introduction of new coverage criteria; its documentation has been improved.
- the implementation has been extended to also handle classical B specifications (e.g, as they are being used in workpackage WP1), in particular return values of operations.
- the Tcl/Tk GUI has been improved (see Fig. 5.1), with more fine-grained selection of events and better feedback.

- the performance of the algorithm has been substantially improved. E.g., the algorithm is now 8 times faster for generating full event coverage for a Volvo Cruise controller model (one of our reference models): the runtime has been reduced from about 80 seconds down to 10 seconds. Further improvements and a combination with the ProB model checker are planned within Advance.

5.2 Evaluation of ProB performance on Setlog Test-case Generation Benchmarks

A series of constraint solving tasks that arose during test-case generation for Z models has been published in [CF12]. The authors have evaluated ProB on a series of benchmarks in [CRF12, CRF13]. While ProB fares much better than setlog on the benchmarks, the authors also present a new extended version of setlog, which fares slightly better than ProB. However, the paper used an older version of ProB and we have rerun those experiments to evaluate the performance of the latest version of our ProB constraint solver. Overall, the performance of ProB has improved and a large number of test-case generation constraints can now be solved by ProB. We describe a selection of those below, to give the reader a flavour of the kind of constraints that arise in test-case generation and can be solved. In principle, the performance of ProB could be further improved by replacing the abundant use of the `STRING` datatype by more focussed enumerated sets. In order to not change the experimental setup from [CRF12, CRF13], we have not done this below. Also note that [CRF12, CRF13] made a mistake in setting the time-out value of ProB; as such the timings appear larger than they actually are.

Below, we want to give the reader a flavor of the kind of constraints that ProB can now solve. We always call ProB with the following parameters:

```
-p BOOL_AS_PREDICATE TRUE -p CLPFD TRUE -p MAXINT 127 -p MININT -128
```

For the experiments below, the constraints for a valid test-case a put into a file. For example, this is the file `bank-Extraer_SP_29.prob` containing constraints for a particular test scenario for a banking system:

```
 cajas : STRING +-> NATURAL &
 num : STRING &
 m : NATURAL
&
```

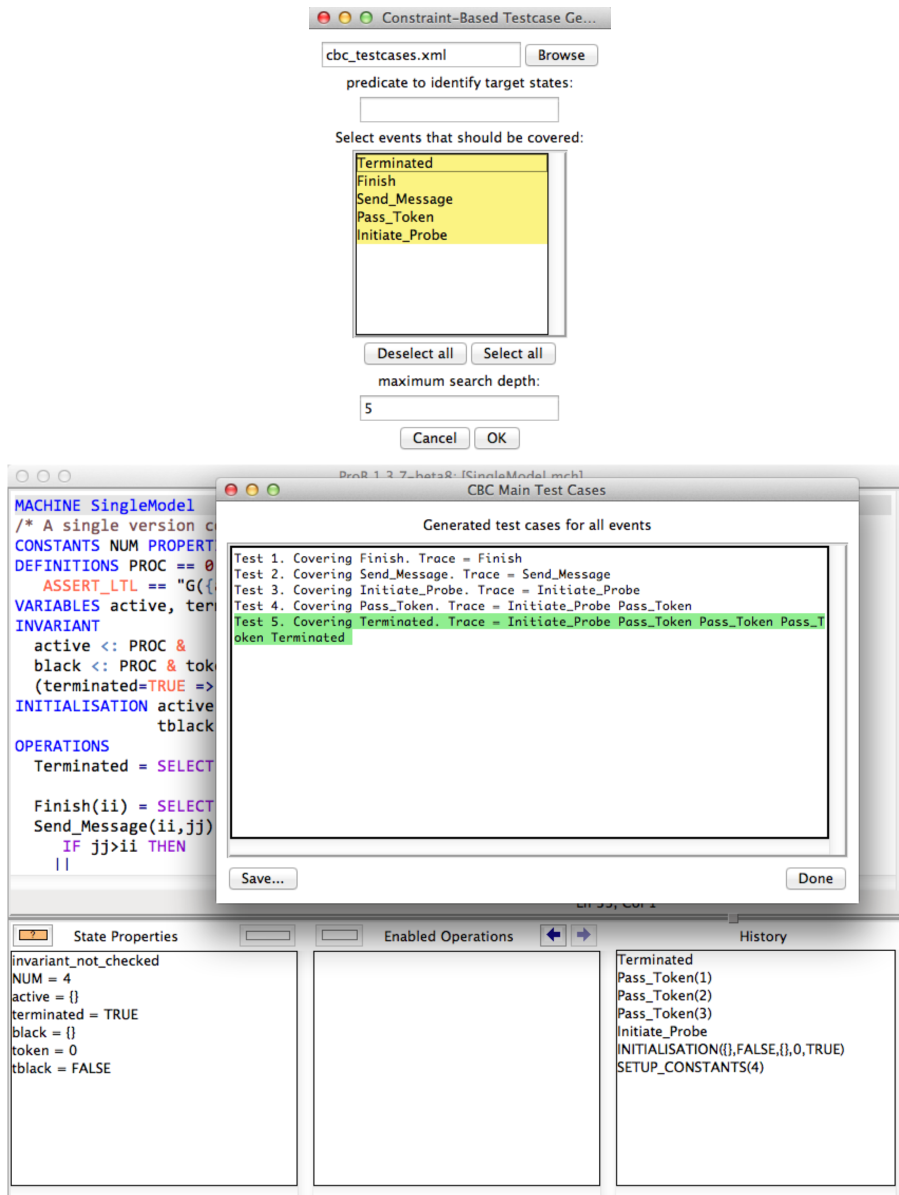


Figure 5.1: The Tcl/Tk Interface to the Constraint-Based Test Case Generator

Procedure CBTestCaseGeneration

Input: The events to cover to_cover,
and the maximum test case length max_length.

$\backslash$— *Initialisation* —$\backslash$

$paths := \{[\,]\}$ — $[\,]$ is the path of length 0

$testcases := \varnothing$

$cur_length := 1$

$\backslash$— *Breadth first search* —$\backslash$

while $cur_length \leq \max_length \wedge to_cover \neq \varnothing$ **do**

$next_paths := \varnothing$

for each path $p \in paths$ and event e **do**

$s_1 :=$ **solve** constraints of the path $p \leftarrow e$
where its last state fulfills the target predicate.

if $s_1 \neq fail$ **then**

$next_paths := next_paths \cup \{p \leftarrow e\}$

$testcases := testcases \cup \{s_1\}$

$\backslash$— *With* $p_1, \ldots, p_{cur_length-1}$ *beeing the events in* p*:* —$\backslash$

$to_cover := to_cover \setminus \{p_1, \ldots, p_{cur_length-1}, e\}$

else

$s_2 :=$ **solve** constraints for the path $p \leftarrow e$

if $s_2 \neq fail$ **then**

$next_paths := next_paths \cup \{p \leftarrow e\}$

end for

$paths := next_paths$

$cur_length := cur_length + 1$

end

Figure 5.2: Constraint based test case generation

```
m > cajas(num) &
num : dom(cajas) &
cajas /= { } &
{ num |-> ( ( cajas(num) ) - m ) } /= { } &
dom({ num |-> ( ( cajas(num) ) - m ) }) <<: dom(cajas)
```

This constraint can be solved with the latest version of ProB; note that several of the variables are unbounded (hence also the enumeration warnings below):

```
 probcli -eval_file bank-Extraer_SP_29.prob
### Warning: enumerating INTEGER : 0:sup ---> 0:127
### Warning: enumerating STRING : inf ---> "STR1","STR2",...
### Warning: enumerating NATURAL(1) : 0:sup ---> 0:127
```

```
Existentially Quantified Predicate over cajas,num,m is TRUE
Solution:
       cajas = {(""STR1""|->0),(""STR2""|->0)} &
       num = ""STR1"" &
       m = 1
```

Another test-case generation constraint that can now be solved is `bancoG-Withdraw_NR_22.prob`:

```
 clients : STRING +-> STRING &
 balances : STRING +-> NATURAL &
 owners : STRING <-> STRING &
 u : STRING &
 n : STRING &
 m : NATURAL
&
 m > balances(n) &
 m < 100000
```

A more complicated benchmark constraint is the following one, `qsee-LoadMemory_SP_11.prob` (quite similar to another one `qsee-LoadMemory_SP_12.prob`):

```
 srv : {"HS","SC","COM","VOM","SDA","RDA","TD","RA","GMR","LDM","EP","MSP"} &
 om : {"SAFETY","NOM","DIAG"} &
 acquiring : {TRUE,FALSE} &
 prepData : {TRUE,FALSE} &
 prepDataType : {"SD","HD","MD"} &
 page : NATURAL &
 ia : NATURAL & fa : NATURAL &
 lpck : seq(STRING) &
 lpckDT : {"CR","CD","OM","ND","ASD","AMD","MSE","ICS","LS"} &
 csrs : {"SD","HD","MD"} --> NATURAL & totCSR : {"SD","HD","MD"} --> NATURAL &
 csc : NATURAL &
 mem : 1 .. 1024 --> STRING &
 modMem : POW(NATURAL) &
 sdwp : NATURAL &
 sparam : {"HGT","ILP","AICST"} --> NATURAL &
 time : NATURAL &
 processingCmd : {TRUE,FALSE} &
 iain : NATURAL &
 data : seq(STRING) &
 c : NATURAL
&
 processingCmd = TRUE &
 srv = "LDM" &
 om = "NOM" &
 csc > 0 &
 c = csc - 1 &
```

```
 sparam("ILP") <= iain &
 iain + card(data) : 0 .. 32 &
 {x | x-iain:dom(data)} /\ modMem = { } &
 {x | x-iain:dom(data)} /= { } &
 modMem = { }

probcli -evalt_file qsee-LoadMemory_SP_11.prob
### Warning: enumerating NATURAL(1) : 0:sup ---> 0:127
### Warning: enumerating INTEGER : 0:sup ---> 0:127
### Warning: enumerating INTEGER : 1:sup ---> 1:127
### Warning: enumerating STRING : inf ---> "STR1","STR2",...
### Warning: enumerating seq (length) : inf ---> 127
Existentially Quantified Predicate over srv,om,acquiring,prepData,
 prepDataType,page,ia,fa,lpck,lpckDT,csrs,totCSR,csc,mem,modMem,
 sdwp,sparam,time,processingCmd,iain,data,c is TRUE
Solution:
       srv = "LDM" &
       om = "NOM" &
       acquiring = TRUE &
       prepData = FALSE &
       prepDataType = "HD" &
       page = 0 &
       ia = 0 &
       fa = 0 &
       lpck = [] &
       lpckDT = "AMD" &
       csrs = {("HD"|->0),("MD"|->0),("SD"|->0)} &
       totCSR = {("HD"|->0),("MD"|->0),("SD"|->0)} &
       csc = 1 &
       mem = #1024:{(1|->""STR1""),(2|->""STR1""),...,
                    (1023|->""STR1""),(1024|->""STR1"")} &
       modMem = {} &
       sdwp = 0 &
       sparam = {("AICST"|->0),("HGT"|->0),("ILP"|->0)} &
       time = 0 &
       processingCmd = TRUE &
       iain = 0 &
       data = [""STR1""] &
       c = 0
```

As you can see, ProB is able to construct large datavalues (such as `mem` above). Finally note, that on no benchmark file of [CRF12, CRF13] did the new version of ProB fare worse than the previous one. All in all, 54 of the 68 testcase benchmarks are solved in 17 seconds using a time-out of 1 second per benchmark (compared to 24 with `setlog` and also 52 with an extended version of `setlog` generated for those benchmarks; the extended version can solve 54 testcases in 13 minutes and 43 seconds). A few more can be solved by ProB if we replace the generic unbounded `STRING` type by more focussed

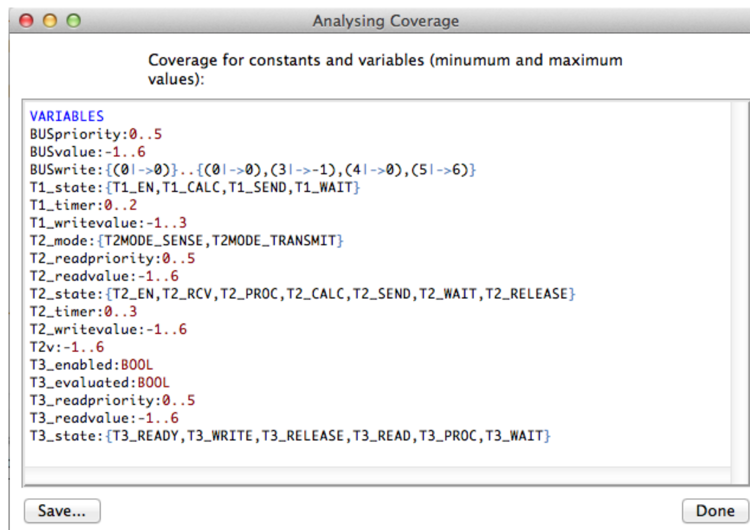


Figure 5.3: Minimum-Maximum Coverage Information Feedback on CAN Bus Hardware model

enumerated types. In summary, this evaluation on an external set of test-case generation benchmarks has shown ProB to perform very well and confirms that the tools has improved considerably over this period of the Advance project.

5.3 Minimum-Maximum Variable Coverage

Based on the needs from the case studies, we have added a new feature to ProB: it can keep track of the minimum and maximum values of variables encountered during model checking or test-case generation.

This feature provides a quick measure of data coverage. It is also useful in detecting unexpected behaviours and identifying why model checking explodes. In one hardware model, this feature allowed us to quickly pinpoint which variable is responsible for an exploding (in this case infinite) state space: by observing the minimum/maximum coverage date (see Figure 5.3) one could see that the `BUSwrite` queue was growing in an unbounded fashion.

Formally, the minumum and maximum values uses an ordering of values $v \prec w$ of type τ, defined inductively as follows:

- if τ=`INTEGER` then $v \prec w$ iff $v < w$
- for τ=`BOOL` we have `FALSE` $\prec$ `TRUE`

- for $\tau = E$ being an enumerated set defined by `E = {e1,e2,...,ek}`, we define `ei` $\prec$ `ej` iff $i < j$.

- for $\tau = D$ being a deferred set instantiated by ProB to `D = {d1,d2,...,dk}`, we define `di` $\prec$ `dj` iff $i < j$.

- for $\tau = \tau_1 \times \tau_2$ we have $(v_1, v_2) \prec (w_1, w_2)$ iff $v_1 \prec w_1 \vee (v_1 = w_1 \wedge v_2 \prec w_2)$

- for $\tau = POW(\tau_1)$ we define $v \prec w$ iff $card(v) \prec card(w) \vee (card(v) = card(w) \wedge max(v) \prec max(w))$, where max makes use of $\prec$ for the smaller type τ_1.

Chapter 6

Constraint Solving

The deliverable also describes the improvements made to the constraint-solving kernel of ProB which is a pre-requisite for constrained random testing.

6.1 General Improvements

Based upon feedback from the industrial case studies, the ProB's kernel has been improved to deal better with large data in particular in conjunction with large or infinite functions. The kernel now detects when infinite types have been enumerated using an exception mechanism. For example, if during the computation of a set comprehension such an exception is raised, then the set comprehension is kept symbolic and not expanded. For example, given the constraint:

```
ev = { x | x:NATURAL1 & x mod 2 = 0 } & 2000:ev
```

ProB now automatically detects that `ev` should be kept symbolic and not be expanded; its membership will be tested upon demand.

We have also enlarged the number of operations that can be applied to symbolic representations compilation first steps.

Many performance improvements have been put into the kernel over the last year. One particular experiment we want to mention is a model from workpackage 1, where ProB is now simulating the B-model of a dynamic interlocking controller about 2.5 times faster than a hand-translated version of the model in OCAML.

6.2 Random Enumeration

The current version of ProB's constraint-solving kernel relies on linear enumeration of the possible values of a variable. This leads to long solving times for constraints in which a solution is only found after several enumeration steps. Furthermore, ProB can only perform a certain number of enumeration steps, before a timeout is reached.

To overcome this limitation, we are experimenting with different enumeration orders. As a first step, we implemented random enumeration of integer variables and booleans. While this has the same worst case complexity as linear enumeration, it might hit a valid solution sooner. Currently, our benchmarks range from small slowdowns to a speedup of two for certain cases. When enabled for the test-case generation benchmarks from Section 5.2, several difficult constraints can now be solved. More benchmarks and a more thorough investigation of the performance impact have to be performed.

6.3 CHR

Aside from evaluating our constraint-solving techniques, we also investigate the use of the CHR - Constraint Handling Rules - language. CHR is a declarative language extension, which can be embedded into languages like Prolog, Java or Haskell. That makes it easily embeddable with the current Prolog core of ProB. CHR is a language especially crafted for writing constraint solvers. It features a compact and easy to understand syntax that might be useful to keep ProB maintainable.

We evaluated the use of CHR in several parts of ProB's core. Furthermore, a prototypical implementation of a constraint solver for boolean conjunction, disjunction and negation was written and benchmarked. While the current prototypes suffer from performance issues, we are confident that further investigation and development in this area is necessary and advantageous.

6.4 ProB as an SMT solver

In order to test and verify the formerly mentioned concepts, we started implementing a parser and translator for constraint satisfaction problems written in the SMT-LIBv2 language. This enables us to use the benchmarks and testfiles collected by the SMT-LIB project. Currently, ProB can evaluate simple problems in the AUFLIA and QF_LIA logics specified in the SMT-LIB standard.

We plan to further improve ProB's ability to read and evaluate SMT-LIBv2 files. Systematically testing the available benchmarks should help to identify persisting bottlenecks in the constraint-solving kernel. Furthermore, the additional set of test cases increases the confidence in ProB.

6.5 Kodkod updates

Previously we have developed a SAT backend for ProB, using the Kodkod library developed by MIT. This can be interesting for solving boolean constraints (e.g., large hardware model) or for constraints involving relational operators (image, relational composition,...) which can be dealt with effectively by Kodkod.

ProB now uses the new version 2.0 of the Kodkod library. We have implemented several small additions (e.g. support for some operators on sequences). Also, several encountered bugs have been fixed. More details can be found at the ProB Bugtracker:
`http://jira.cobra.cs.uni-duesseldorf.de/`.

Bibliography

[CF12] Maximiliano Cristiá and Claudia S. Frydman. Extending the test template framework to deal with axiomatic descriptions, quantifiers and set comprehensions. In John Derrick, John A. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Proceedings ABZ'2012*, pages 280–293, 2012.

[Col12] J. Colley. ADVANCE Deliverable D1.2, Proof of Concept Application in Railway Domain. Technical report, 2012.

[CRF12] Maximiliano Cristiá, Gianfanco Rossi, and Claudia S. Frydman. {log} as a test case generator for the test template framework. Technical report, December 2012. N. 508.

[CRF13] Maximiliano Cristiá, Gianfranco Rossi, and Claudia S. Frydman. {log} as a test case generator for the test template framework. In Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti, editors, *SEFM*, volume 8137 of *Lecture Notes in Computer Science*, pages 229–243. Springer, 2013.

[LSBL08] Michael Leuschel, Mireille Samia, Jens Bendisposto, and Li Luo. Easy graphical animation and formula viewing for teaching B. In C. AttiogbÃľ and H. Habrias, editors, *The B Method: from Research to Teaching*, pages 17–32. Lina, 2008.

[MBER10] I. Maamria, M. Butler, A. Edmunds, and A. Rezazadeh. On an Extensible Rule-based Prover for Event-B. In *ABZ2010*, February 2010.

[MOD10] MODELISAR. Functional Mock-up Interface for Co-Simulation, Version 1.0. `https://svn.modelica.org/fmi/branches/public/specifications/FMI_for_CoSimulation_v1.0.pdf`, October 2010.

[PL07] Daniel Plagge and Michael Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods*, volume 4591 of *Lecture Notes in Computer Science*, pages 480–500. Springer-Verlag, 2007.