D.4.4 - Methods and Tools for Simulation and Testing III

ADVANCE

Grant Agreement:	*287563*
Date:	*30/11/2014*
Pages:	*32*
Status:	*Final*
Authors:	*Michael Leuschel University of Düsseldorf*
Reference:	*D4.4*
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 D.4.4

Method and tools for simulation and testing III

Public Document

November 30th, 2014

http://www.advance-ict.eu

ADVANCE Deliverable D4.4
Method and tools for simulation and testing III

Deadline: 31st October; review by November 10, Final Version due by end of November
Contributors: Jens Bendisposto, Joy Clark, Ivaylo Dobrikov, Dominik Hansen, Lukas Ladenberger, Michael Leuschel, Aymerick Savary, Vitaly Savicks, Harald Wiegard, John Witulski
Reviewers: Jose Reis

D4.4) Methods and tools for simulation and testing III: This deliverable covers the final version of all simulation, code generation and testing tools. It describes the extended Model-Based Testing framework and the associated documentation and tutorials provided to support the new constrained random testing method. The supporting coverage metrics and coverage detection and reporting mechanisms are also described. Finally, the results of the code generation development are described and measured against the plan. Documentation and tutorials for the generation of both Ada and C models will be provided, and the simulation efficiency of these generated models measured. [month 36]

Summary and Overview

During the lifetime of the project, major enhancements and new techniques and tools have been developed around simulation, multi-simulation, code generation, constraint solving and and testing.

A lot of the work in this workpackage has centred around the **ProB** toolset. Within Advance, we have developed a new ProB Java API (Application Programming Interface), which provides a clean, scriptable API for the ProB (multi-)simulation, but also model checking and constraint solving features. While requiring substantial development effort, the ProB Java API has provided enormous benefit. The ProB Java API has been used extensively for workpackages WP1 and WP2, but is also starting to be used extensively outside of Advance. We hope the ProB Java API will provide a lasting impact for the Advance developments. Below, we also describe several such uses, such as a University course planner (SlotTool), integration into the TLA+ Toolbox, or the iTool interface to make the results of this project available to other communities (in particular VDM with Ouverture or ASMs).

During the Advance project, a new version of the ProB **visualization** editor **BMotion Studio** has been developed, called BMotion Studio (for ProB) 2. BMotion Studio 2 uses web technologies like SVG, HTML, CSS and JavaScript as the underlying graphical engine. This enables the user to create domain specific visualisations and to reuse existing components found in the world wide web, like third party JavaScript libraries and SVG graphics. BMotion Studio 2 can make use of the entire ProB Java API for creating complex and dynamic visualisations. In particular, the scripting support of BMotion Studio 2 has been used extensively for creating visualisations of workpackages WP1 (see Figure BMS_2) and WP2 (see Figure BMS_3 and BMS_4).

Multi-Simulation is a core technology for the ADVANCE project. During the course of the project, the decision was made to build the ADVANCE multi-simulation tools upon an industry standard, namely the FMI Interface. In a first instance, an FMI layer was added to ProB and the ProB Java API. This enables one to develop multi-simulation masters programmatically in Java or Groovy, which can coordinate FMI components (called FMUs) and Event-B machines simulated by ProB.
For a more user-friendly interface to the simulation-based validation, a new **MultiSim** multi-simulation plug-in for Rodin has been developed, which enables composition and simulation of multiple Event-B machines and physical models of environment. The plug-in supports importing any physical models compliant with the FMI 1.0 for Multi-Simulation standard and provides a graphical composition editor and simulation environment. The simulation is performed by a generic master algorithm that utilises the developed interface to FMI models and the ProB Java API for the simulation of Event-B models.

At the heart of the ProB tool lies the **constraint solver**. It is particularly important for model-based testing, but also for effective simulation of very high-level models. During the lifetime of the ADVANCE project, major performance improvements have been achieved on this core part of ProB (see also the benchmark results shown below and in the various ADVANCE publications). The capabilities of the solver have also been considerably improved: it can deal much better with symbolic and potentially infinite sets. ProB now also automatically detects those sets which it cannot represent finitely and keeps them symbolic. It is tthe only solver we know of that is capable of dealing with **large** integers and data values, automatically detecting potentially **infinite** sets and attempting to solve constraints over **higher-order** data structures. Another main contribution to ADVANCE was the support for Event-B theories, which greatly relies on being able to deal with symbolic sets and functions. It is also now possible to perform random enumeration and the use of ProB as a prover. We have also developed an alternative SAT-Solving backend for ProB, based on the Kodkod library.

Within the project, we have also advance the state of the **model-based testing** technology for Event-B. The performance of the constraint-based test case generation has been considerably improved (see some numbers in the model-based testing section below), and the algorithm has been made more intelligent by using statically computed enabling and feasibility information. The features of the model-checking based testcase algorithms have also been extended. All of this has also lead to considerable use of this technology outside of the ADVANCE project. Finally, we have implemented the MC/DC coverage criterion, which has proven to be useful not just for testcase generation but also for the static analysis of models (e.g., to detect vacuous guards and invariants).

Work on **code generation** had begun in previous projects, but the tool was a ‘proof-of-concept’ prototype. In ADVANCE, the code generation tools matured, existing plug-ins were upgraded to improve usability, and the tool was extended to generate code from state-machines. For co-simulation with FMUs, a feature for generating code from component diagrams was added. Various research tracks were followed, including instrumenting code to vary simulations; templates for code reuse, configuration, and code-injection; and exploratory work on translation using complex data types.

In summary, considerable advances to the state of the art have been made by the ADVANCE project in the area of multi-simulation, constraint solving and model based testing. The applications of this technology has been very successful within workpackages WP1 and WP2. In addition, the technology has already had considerable impact outside of ADVANCE, leading to numerous external applications of the technology.

In the following sections of this deliverable we describe the major enhancements and new developments conducted during the last period of the project.

Links to tools

A major part of D4.4 deliverable are the final version of the tools, which are available at the following web sites:

- ProB Advance Release http://www.stups.uni-duesseldorf.de/ProB/index.php5/prob2-advance-release
    - ProB http://www.stups.hhu.de/ProB/
    - ProB Java API http://www.stups.hhu.de/ProB/index.php5/ProB_Java_API
    - BMotion Studio http://www.stups.hhu.de/ProB/index.php5/BMotion_Studio
- Multi-Simulation Plug-in (Multi-Sim) Git repository: https://github.com/snursmumrik/rms2/tree/multisim
- Code Generation: Code generation Wiki Page

Links to the tools are also maintaned at: http://www.advance-ict.eu/tools

Updates in last period

This section of the deliverable shows the major updates in the last period with links to articles, technical reports and more extensive documentation. A lot of resources went into ProB (the kernel, the Java API and the BMotion Studio visualization engine and editor), driven by the (heavy) demands of the case studies. These have taken on in complexity and challenge during the final period of ADVANCE. As such, support, stability and performance improvements were very important. An executive summary of the main achievements has been provided in the introduction above; many of these achievements have been finalised in the last period.

ProB Java API

Motivations / Decisions

The ProB Java API was conceived as an attempt to make it easier to integrate the ProB tool into existing applications. Because the Java language is so widely used, particularly when developing tools for the Eclipse-based Rodin project, encapsulating the power of ProB in a Java API has already made it simple to integrate ProB into existing Java applications and to create new applications that utilize the functionality of ProB. Because of the integration of the Groovy programming language, a dynamic language for the JVM, animations can be controlled via scripting, and a REPL is available for the user to interact with the Java core.

Improvements to ProB Java API

Performance Improvements

The WP1 and WP2 workpackages have used the ProB Java API extensively in their case studies in order to perform animation and visualization with BMotion Studio 2.
In the original implementation of the Java API, a naive and not necessarily efficient approach was often used to solve certain issues. At the time, this was not an issue, because the goal was to create a functioning and useful API as quickly as possible. However, when using the existing API, the case studies revealed the following performance issues mainly caused by high memory consumption and unnecessary communication between Java and Prolog:

Performance issues found during WP1 case study

While creating a visualization for the Malaga model for WP1, we determined that one of the main performance issue that faced the new Java API was the communication between the Prolog CLI binary and the Java API.

The Java API communicates with the ProB CLI via socket, and the results from the CLI are in a string representation that must be parsed when they are received by the Java application. We determined that two factors drastically slow down the Java API: the frequency with which the CLI is called and the size of the Strings that are sent between the two applications.

In order to solve the first issue, we have worked to reduce the number of communication steps that take place by identifying where multiple communication steps can be made in one step and then consolidating those steps into one call to the CLI.

After reading the environment values the controller makes its decision by executing several events multiple times, i.e. it executes the first event of the decision procedure until it cannot be executed then it runs the second command in the cycle. Because we do not know how many times a single event can be called, we could not combine the calls into a single invocation on the Prolog part of ProB. Instead, we implemented a new command that executes an event until it either becomes disabled or (optionally) a specific number of invocations was reached. Using this command, ProB can execute the decision cycle in Prolog without communicating with Java.

We have worked at reducing the size of the strings by identifying calls that return redundant or unnecessary information back to the Java API and eliminating the unwanted information. For example, for the Malaga model, the parameters for a given event can be extremely large. After each animation step, all the calculated parameters would be sent to the Java API even though the value of the parameters was not needed for the visualization. We restructured the API so that the parameters would be lazily retrieved from the CLI when needed and not sent after every communication step.

The resulting visualization is about twice as fast as the original version.

Performance issues found during WP2 case study

For the WP2 mesh network, a trace created via multi-simulation to represent the behavior of the system over the course of a day consists of roughly 100,000 events. Trying to replay this trace on a computer with normal computing capabilities would lead to a heap overflow error because of a naive and inefficient implementation of the base classes used in the API, because of an excess of duplicate objects, and because of the holding of references to unused objects that prevented the garbage collector from working. By doing memory profiling with the Trace files from the case study and identifying the problem classes, we were able to reduce the memory footprint enough that it is possible to easily replay a Trace with over 160,000 events without exceeding 800MB of memory, which is reasonable for using the tool on computers with a normal computing capacity.

A second improvement was the reduction of communication between Java and Prolog. The multi-simulation process involves reading input values from a continuous model representing the environment and then performing discrete animation steps until a certain event is observed. At this point, the values of the discrete model are returned to the continuous model which calculates new input values for the discrete model. Each of these cycles contains roughly 150 animation steps. In the original animation, each animation step was randomly executed from the Java side until the synchronization event appeared. This meant that for one simulation cycle, about 150 communication steps were needed between the Java API and the Prolog binary. As a solution to this problem, we implemented a feature in the Prolog CLI that allows random animation to occur until an LTL condition has been met. Since the observation of an event can be encoded as an LTL condition, this enabled us to perform all the random animation in the Prolog CLI and reduced the communication steps needed for a simulation cycle from around 150 to two: one to execute the event to read the inputs from the environment into the discrete model and one to perform random animation until the model is ready for new inputs.

The original visualization, in which each event was individually observed, took around three hours. By only using the inputs from the environment and the new random animation command, we reduced the time for the visualization to less than a half an hour, which is a factor 6 faster than the original version.

UI Improvements

The ProB Java API has been successful at integrating ProB into Java applications. The API also provides UI components based on web technologies that can be embedded into any UI framework that contains a browser component. The UI interface is able to track multiple animations at any given time. Each UI component listens for and reacts to changes in the current animation which are broadcast as a trace composed of all of the events that have been executed over the course of the animation. One view shows all of the animations that are currently being tracked by the ProB API. The user can use this view to switch between animations. The Java API also allows users to generate traces based on certain criteria (e.g. a trace to a state for which a user-defined predicate holds). When a trace has been generated, these can be simply added to the user interface so that the user can inspect them further. One view that has been developed is a view to provide the user with access to the model checking capabilities within ProB. In this view, the user can begin model checking jobs using the different model checkers available in ProB (i.e. consistency checking, constraint based checking, LTL model checking). If an error is found via model checking, a trace to the error state is generated and can be added from the model checking view into the UI so that it can be inspected in more detail.

ProB Java API Integration / Related External Projects

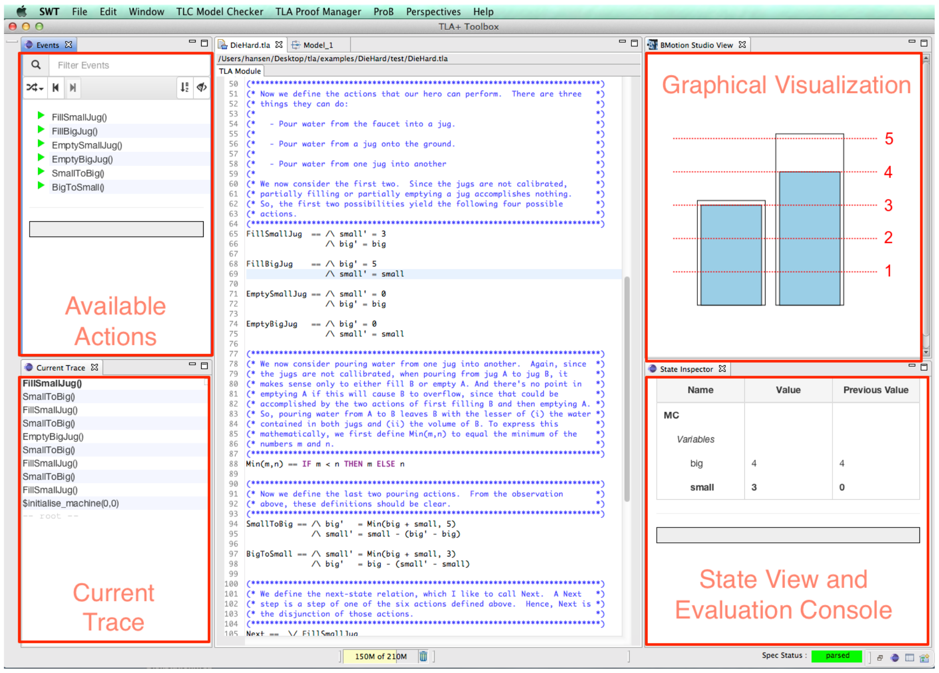
Rodin 3 Integration

Because the UI components provided by the ProB Java API are based on web technologies, we were able create a simple plugin for the Rodin 3 tool that provides the user with all of the functionality of ProB within Rodin. The plugin uses views with embedded Eclipse SWT browsers to access the user interface components that are shipped in the Java API library.

TLA Toolbox Integration

In order to provide a convenient way to use ProB as new validation tool for TLA+, we integrated the ProB Java API into the TLA toolbox. The TLA+ plug-in reuses large parts of the ProB Java

API plug-in for Rodin. We only had to develop a small plug-in that contains the UI bindings for the toolbox and the code for loading TLA+ models.



```
(***************************************************************************)
(* Now we define the actions that our hero can perform.  There are three   *)
(* things they can do:                                                     *)
(*                                                                         *)
(*   - Pour water from the faucet into a jug.                              *)
(*                                                                         *)
(*   - Pour water from a jug onto the ground.                              *)
(*                                                                         *)
(*   - Pour water from one jug into another                                *)
(*                                                                         *)
(* We now consider the first two.  Since the jugs are not calibrated,      *)
(* partially filling or partially emptying a jug accomplishes nothing.     *)
(* So, the first two possibilities yield the following four possible       *)
(* actions.                                                                *)
(***************************************************************************)
FillSmallJug  == /\ small' = 3
                 /\ big' = big

FillBigJug    == /\ big' = 5
                 /\ small' = small

EmptySmallJug == /\ small' = 0
                 /\ big' = big

EmptyBigJug   == /\ big' = 0
                 /\ small' = small

(***************************************************************************)
(* We now consider pouring water from one jug into another.  Again, since  *)
(* the jugs are not callibrated, when pouring from jug A to jug B, it      *)
(* makes sense only to either fill B or empty A. And there's no point in   *)
(* emptying A if this will cause B to overflow, since that could be        *)
(* accomplished by the two actions of first filling B and then emptying A. *)
(* So, pouring water from A to B leaves B with the lesser of (i) the water *)
(* contained in both jugs and (ii) the volume of B. To express this        *)
(* mathematically, we first define Min(m,n) to equal the minimum of the    *)
(* numbers m and n.                                                        *)
(***************************************************************************)
Min(m,n) == IF m < n THEN m ELSE n

(***************************************************************************)
(* Now we define the last two pouring actions.  From the observation       *)
(* above, these definitions should be clear.                               *)
(***************************************************************************)
SmallToBig == /\ big'   = Min(big + small, 5)
              /\ small' = small - (big' - big)

BigToSmall == /\ small' = Min(big + small, 3)
              /\ big'   = big - (small' - small)

(***************************************************************************)
(* We define the next-state relation, which I like to call Next.  A Next   *)
(* step is a step of one of the six actions defined above.  Hence, Next is *)
(* the disjunction of those actions.                                       *)
(***************************************************************************)
Next ==  \/ FillSmallJug
```

Name	Value	Previous Value
MC		
Variables		
big	4	4
small	3	0

The Toolbox Integration was presented at the TLA+ Community Event (http://tla2014.loria.fr/) and an extended abstract is online available [1].

ABZ Landing Gear Case Study

We have developed a formalisation of the ABZ landing gear case study in Event-B [2]. The development was carried out using the Rodin platform and mainly used superposition refinement to structure the specification. To validate the model we combined proving with animation and model checking. For the latter, we used the ProB animator and model checker. Graphical representation of the model turned out to be crucial in the development and validation of the model; this was achieved using BMotion Studio 2 and the ProB Java API. Figure [ABZ_Landing] shows the visualisation of the Landing Gear developed with BMotion Studio 2.

The main goals of taking part in the ABZ Landing Gear Case Study were to increase the impact and visibility of the ADVANCE technology and to try out and improve the ADVANCE tool and methodology. As a result, many improvements like new features in the LTL model checker and new visualisation features have been developed. For instance, in BMotion Studio 2 a visualisation may be created for different refinement steps. This allows some of the complexity of the system to be hidden and makes it possible to focus on a specific problem.

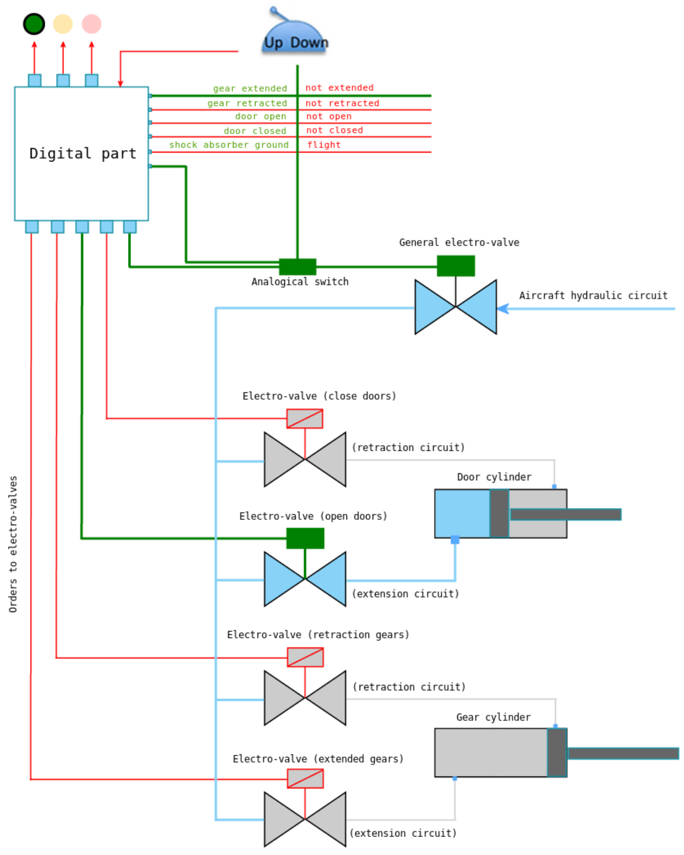
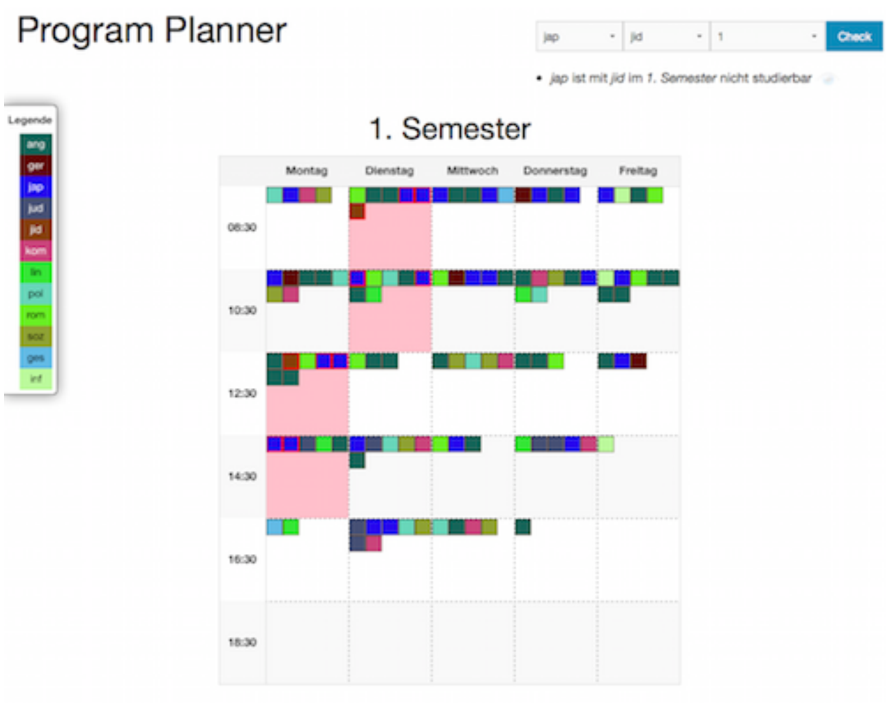


Figure [ABZ_Landing] The ABZ Landing Gear Visualisation

SlotTool Case Study

The ProB Java API has been used as part of an external project with the goal to assist those responsible of creating student timetables at different faculties of the university of Düsseldorf. In this context, we use ProB's constraint solving features to verify properties that should be valid for the published timetables. A web-based user interface is being created in order to edit and validate the timetables. This user interface communicates with a server component, which embeds the ProB Java API as a library to provide a REST based interface for the frontend to interact with the model.



- *jap* ist mit *jid* im *1. Semester* nicht studierbar

Conclusion

By developing the ProB Java API we have been successful in creating an API that enables ProB to be integrated into existing Java applications and also allows new tools that are interested in harnessing the power of ProB to be quickly be developed. Over the course of the Advance project, the ProB Java API has been successfully integrated in several different case studies and projects. Feedback from the industrial case studies in the Advance project has enabled us to identify and fix certain performance issues to improve the overall API. We have also written comprehensive documentation to enable new developers and users to quickly become comfortable with the API.

Available Documentation

- **ProB Java API Developer Documentation:** For developers who want to build specialized tools on top of ProB, we have prepared comprehensive documentation of the ProB Java API together with a template that enables a quick start. The documentation is available from http://nightly.cobra.cs.uni-duesseldorf.de/prob2/developer-documentation/prob-devel.pdf, the template is located at https://github.com/bendisposto/prob2_tooling_template.
- We have also collected additional resources about developing tools on top of ProB here: http://www.stups.uni-duesseldorf.de/ProB/index.php5/ProB_Java_API
- **ABZ Case Study Material:** Links, video and an online animation/visualization of model is available at: http://www.stups.uni-duesseldorf.de/ProB/index.php5/ABZ14

References

[1] Dominik Hansen, Jens Bendisposto, Michael Leuschel. Integrating ProB into the TLA Toolbox. In TLA Workshop, 2014.
http://www.stups.uni-duesseldorf.de/w/Special:Publication/tla_abz14
[2] Dominik Hansen, Lukas Ladenberger, Harald Wiegard, Jens Bendisposto, Michael Leuschel, Validation of the ABZ Landing Gear System using ProB In ABZ 2014: The Landing Gear Case Study, 2014. http://www.stups.uni-duesseldorf.de/w/Special:Publication/abz14casestudy

BMotion Studio 2

Motivations / Decisions

Originally BMotion Studio was developed as a separate plug-in for Rodin [1]. It used the Eclipse Graphical Editing Framework (GEF) to provide a visual editor for creating domain specific visualisations of Event-B models. While this was a very convenient and fast approach to create simple visualisations, the visual editor makes it difficult to create complex visualisations. In particular, the visual editor is limited in terms of design flexibility and very inefficient when creating dynamic visualisations with numerous or repeated elements. For instance, creating a railroad track layout (needed for WP1) or a communication network with several nodes (needed for WP2) would be very cumbersome.

Moreover, the industrial partners had several feature requests for the WP1 and WP2 case studies during the Advance project. For example, they wanted the option to automatically playback a particular trace and to record the visualisation over a simulation period so that it could be played back at later times. The WP2 case study needed to be able to visualise a high number of events (around 300,000) to simulate a full day of the simulation. However, the original BMotion Studio is limited in terms of extensibility and efficiency. Adding most of the desired custom features was not possible without modifying the core code of BMotion Studio. For this reason, a new version of BMotion Studio has been developed during the Advance Project called BMotion Studio 2. One of the main differences between the original version for Rodin and BMotion Studio 2 is that the latter is no longer based on Eclipse but rather uses standard Browser technology as its GUI.

Figure BMS_1 shows an overview of the architecture of BMotion Studio 2. In BMotion Studio 2, a visualisation is described by a *visualisation template* that contains *visual elements* and *observers*. Visual elements may be, for instance, shapes or images that represent some aspects of the model. For example, when modelling a communication protocol, circles may be used for representing the communicating entities of the protocol and arrows for the message exchanges between the entities. BMotion Studio 2 supports web technologies like Scalable Vector Graphics (SVG) [3] and Cascading Style Sheets (CSS) [4] for this purpose. SVG is an XML-based markup language for describing two-dimensional vector graphics. It comes with a number of visual elements like shapes, images and paths. CSS is a language that can be used to describe the style of SVG visual elements (e.g. the colour or the dimension).

Observers are used to link visual elements with the model. An observer is notified whenever a model has changed its state, i.e. whenever an event has been executed. In response, the observer will query the model's state and triggers actions on the linked visual elements in respect to the new state. BMotion Studio 2 comes with a number of default observers for creating visualisations for Event-B. For instance, BMotion Studio 2 provides an observer that takes a user-defined predicate that is to be evaluated in every state. Depending on the result of the predicate (true or false), the observer will trigger an action to change the appearance of the linked visual elements (e.g. the colour of a shape).

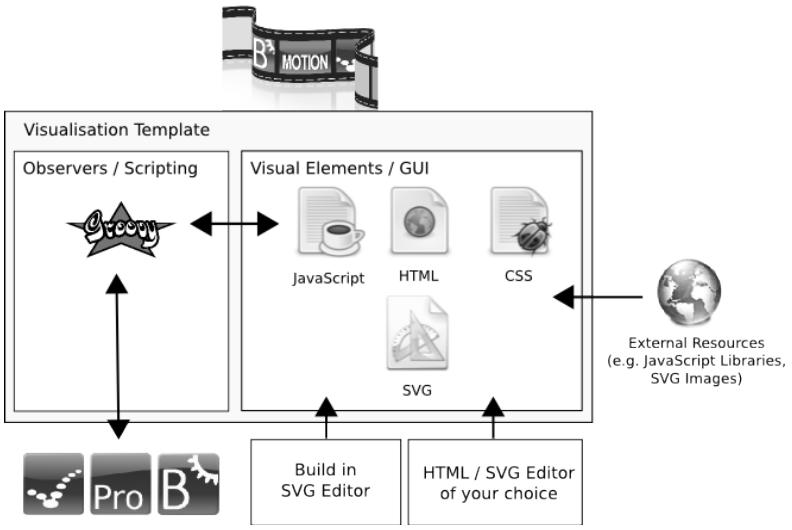


Fig. BMS_1. Architecture of BMotion Studio 2

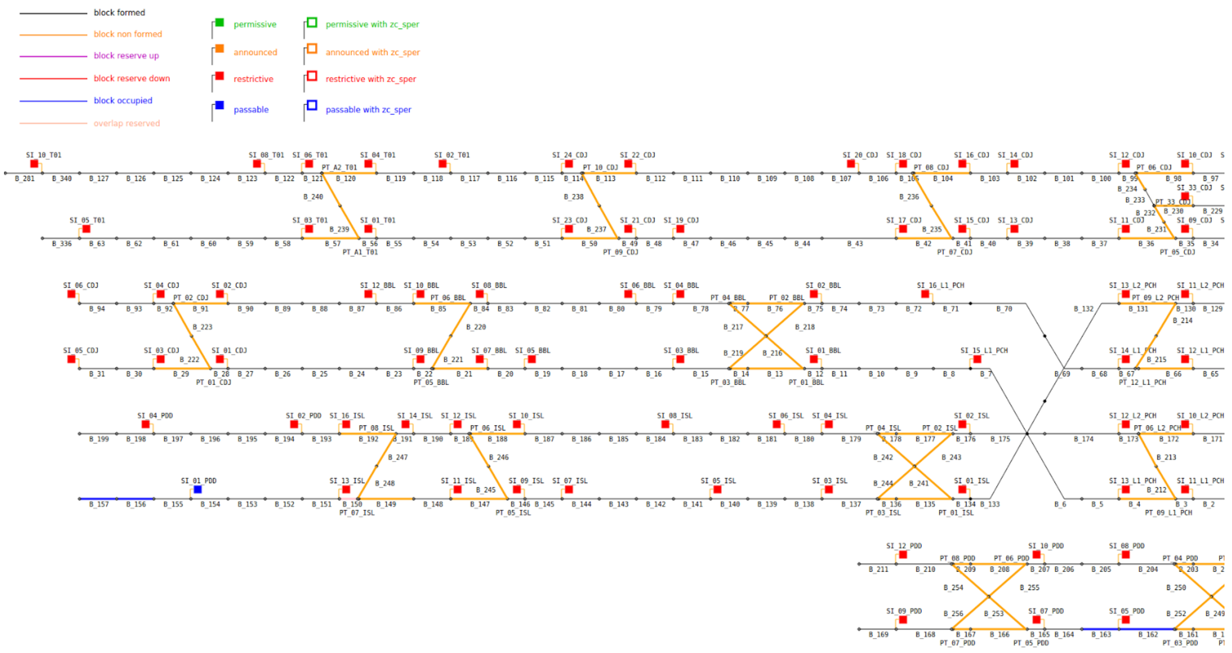


Figure BMS_2. WP1 Visualisation: Interlocking System

Scripting Support

BMotion Studio 2 follows the same principles as the ProB Java API. Most parts of BMotion Studio 2 are accessible via the Groovy scripting language. All of the features in ProB, from the constraint solver to the user interface, are available when writing a Groovy script and are therefore also available to the user when creating visualisations. This enables the user to programmatically control ProB and to access the actual formal model being visualised. In fact, the use of a scripting language facilitates the creation of complex and dynamic visualisations. A track layout, for example, can be created dynamically based on information (e.g. variables and constants) that comes from the formal model being visualised.

The GUI of a visualisation can also make use of JavaScript. JavaScript is a dynamic scripting language which is supported in almost all web browsers. There are a lot of libraries for JavaScript that users can apply to create custom visualisations if they have specific needs in a particular domain. For instance, libraries exist for manipulating the DOM of an HTML document, or for generating chart and plot diagrams.

The scripting support is used exhaustively by the WP1 and WP2 case studies. For instance, a library is used for WP2 to create chart diagrams that show the voltage over time of the selected point in the network and the tap position at the transformer (see Figure BMS_4). The scripting support also enhances the interactivity of the visualisation: The user can select different points in the network during the visualisation and the proper diagram will be updated to reflect the data at that location. Moreover, the user can switch between to different views (Figure BMS_3 and BMS_4) that show different details of the system: a communication network and a low voltage network.

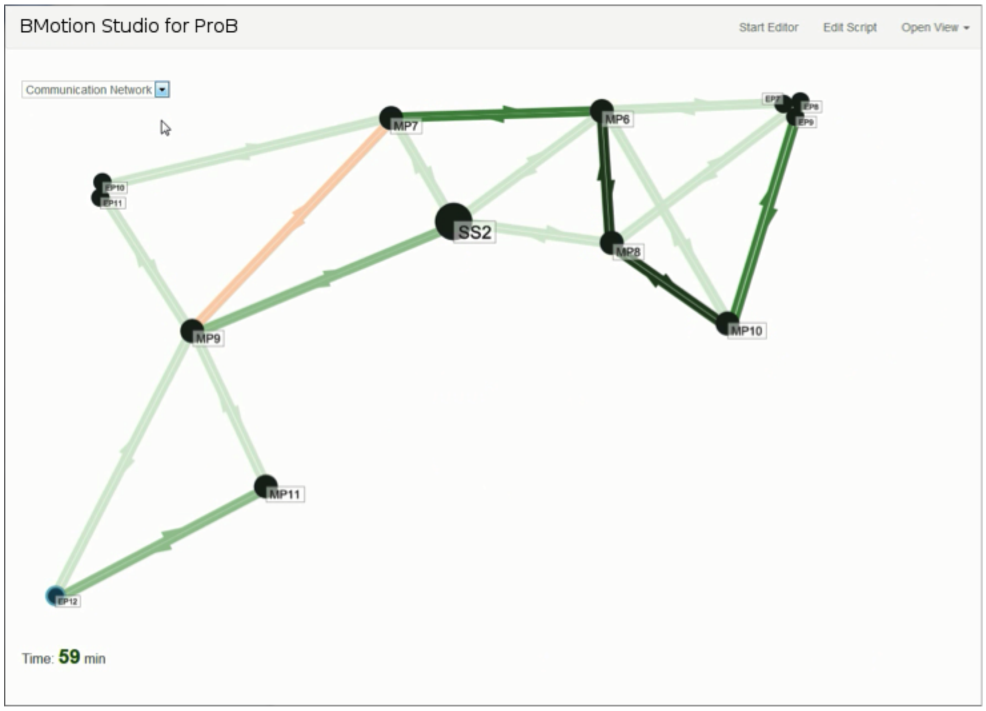


Figure BMS_3. WP2 Visualisation: View of Communication Network

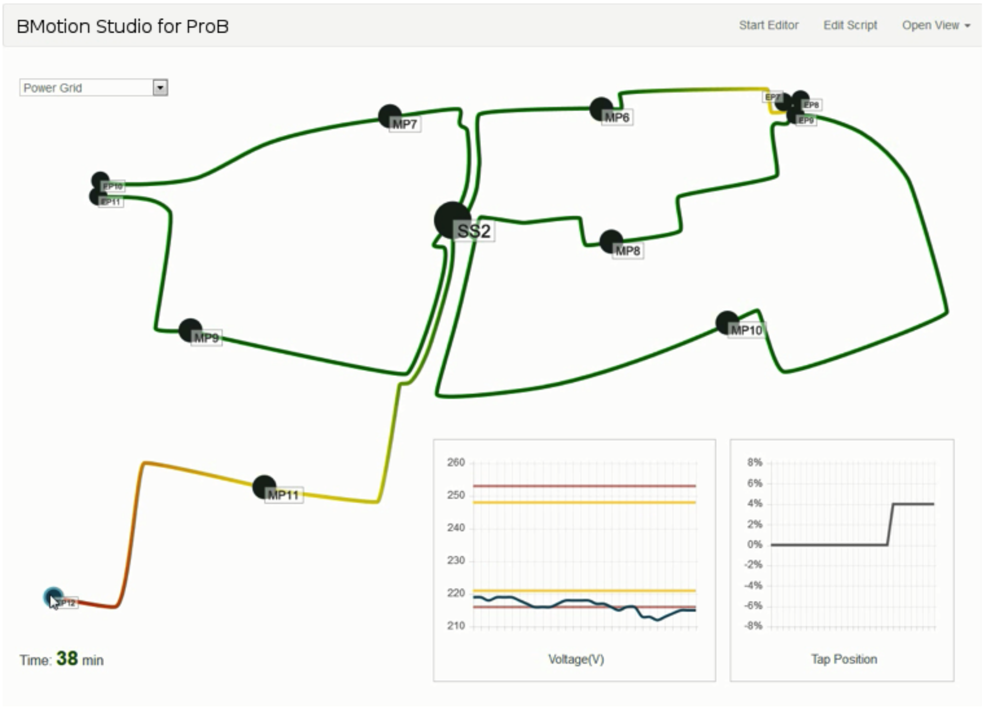
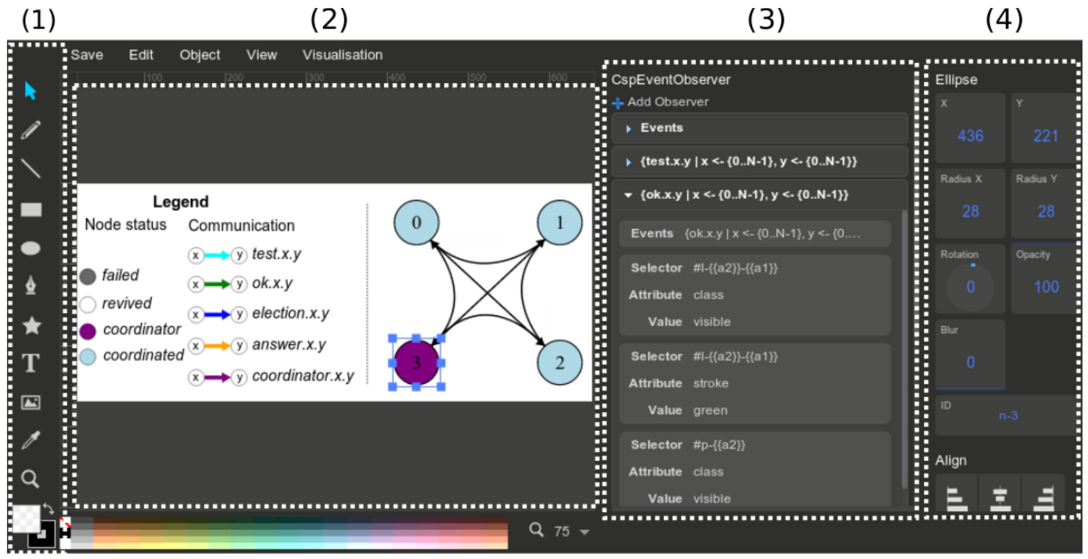


Figure BMS_4. WP2 Visualisation: View of Low Voltage Network

The scripting support also provides a way to automatically playback a specific trace. This feature is used in both case studies, where the visualisation automatically executes events coming from an external log file.

Graphical Editor

BMotion Studio 2 provides a graphical editor with different views and wizards that support users in creating visualisations for formal models. Figure BMS_5 shows the graphical editor with its different views.



CspEventObserver

Events

{test.x.y | x <- {0..N-1}, y <- {0..N-1}}

{ok.x.y | x <- {0..N-1}, y <- {0..N-1}}

Events {ok.x.y | x <- {0..N-1}, y <- {0....

Selector #l-{{a2}}-{{a1}}
Attribute class
Value visible

Selector #l-{{a2}}-{{a1}}
Attribute stroke
Value green

Selector #p-{{a2}}
Attribute class
Value visible

Ellipse

X 436
Y 221
Radius X 28
Radius Y 28
Rotation 0
Opacity 100
Blur 0
ID n-3

Figure BMS_5. BMotion Studio 2 Graphical Editor

The editor consists of a set of tools (1) for creating SVG widgets (e.g. visual elements as shapes and images), a canvas (2) holding the actual visual elements, a view (3) for editing observers, and another view (4) for manipulating the attributes of the currently selected visual element in the canvas.

Although BMotion Studio 2 provides a graphical editor, the user is not restricted to this editor. The user can also make use of any tool that supports the creation of SVG graphics or HTML documents. In case of the WP2 case study, the Inkscape tool has been used to create the SVG graphics.

ITool Interface

In order to adapt BMotion Studio 2 to other simulators and formal languages, a simple interface called ITool has been developed during the Advance Project. As a first case study, the ITool interface has been successfully applied to support the visualisation of CSP-M models (See Section “CSP Visualisation Support”). Moreover, first experiments towards supporting the CoreASM simulator and formal language have been made.

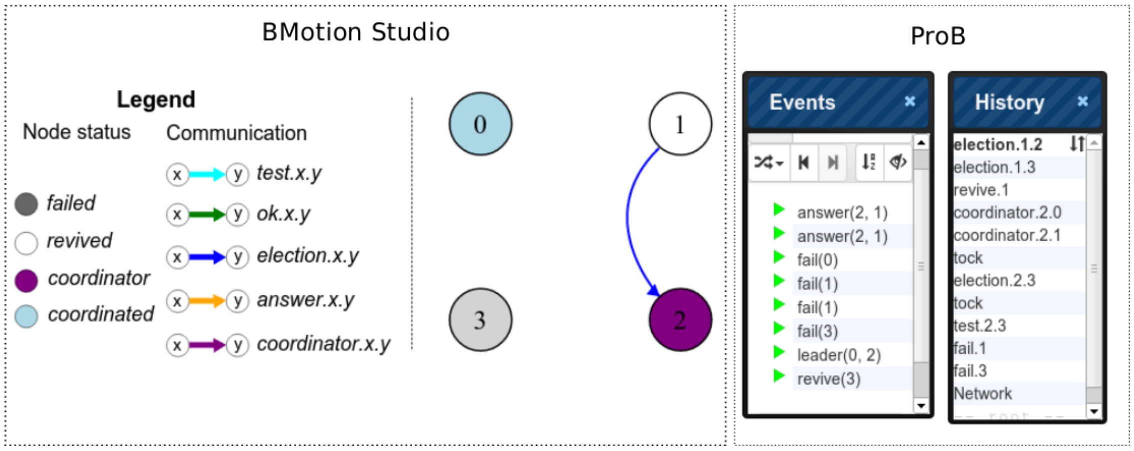
CSP Visualisation Support

Inspired by the successful application of domain specific visualisations of Event-B models, an approach for creating domain specific visualisations for CSP (Communicating Sequential Processes) has been developed [2]. CSP is a notation used mainly for describing concurrent and distributed systems.

The difference between the original visualisation approach for Event-B models and the CSP visualisation approach is imposed by the specifics of the CSP formal language. The basic idea of the Event-B visualisation approach is to visualise the information that is encoded in the states of a model (e.g. the values of variables), where each state of the model is mapped to a particular visualisation. In contrast to Event-B, in CSP the states of the modelled system are left uninterpreted and the behaviour is defined in terms of sequences of events (*traces*). Thus, the concepts of the Event-B visualisation approach are not longer applicable on event-based formalisms as CSP.

The intention of the CSP visualisation approach is to visualise the traces of the underlying CSP model. A trace is a sequence of events performed by a process that can communicate and interact with other processes within the CSP model.

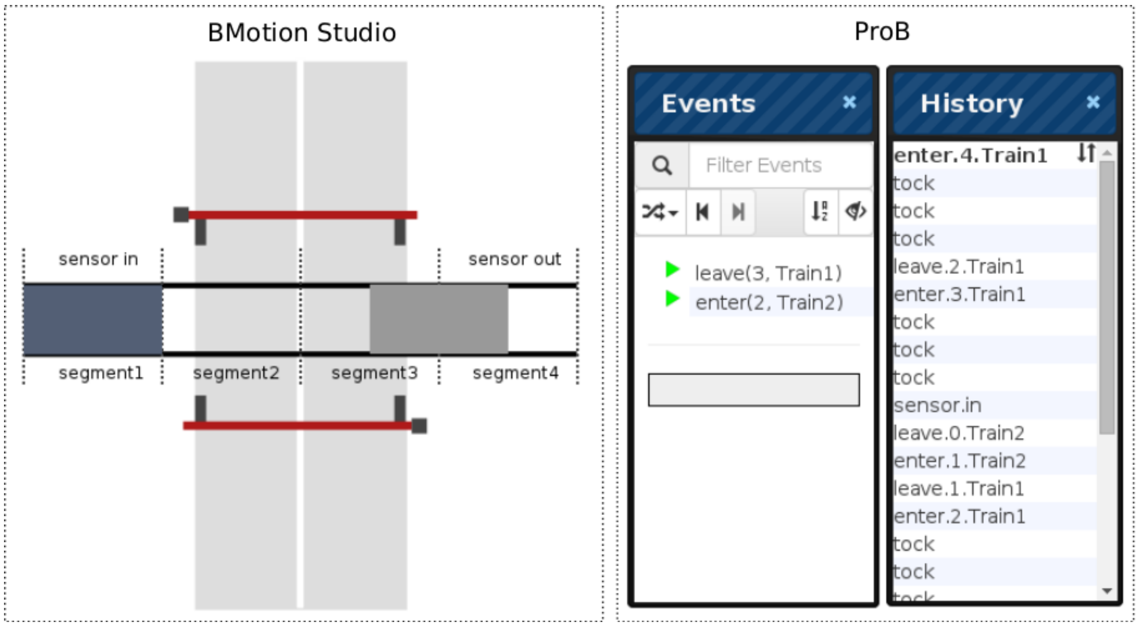
The CSP visualisation approach has been implemented as an extension for BMotion Studio 2. In addition, various visualisations of CSP models have been created in order to demonstrate the approach. For example, Figure BMS_6 and BMS_7 show the visualisation of the bully algorithm specification from [5] and of the level crossing gate specification from [6].



```
answer(2, 1)
answer(2, 1)
fail(0)
fail(1)
fail(1)
fail(3)
leader(0, 2)
revive(3)
```

```
election.1.2
election.1.3
revive.1
coordinator.2.0
coordinator.2.1
tock
election.2.3
tock
test.2.3
fail.1
fail.3
Network
-- root --
```

Figure BMS_6. CSP Visualisation of the Bully Algorithm



```
leave(3, Train1)
enter(2, Train2)
```

```
enter.4.Train1
tock
tock
tock
leave.2.Train1
enter.3.Train1
tock
tock
tock
sensor.in
leave.0.Train2
enter.1.Train2
leave.1.Train1
enter.2.Train1
tock
tock
```

Figure BMS_7. CSP Visualisation of the Level Crossing Gate

Available Documentation

BMotion Studio Website. http://www.stups.hhu.de/ProB/index.php5/BMotion_Studio
ITool Documentation: http://www.stups.hhu.de/ProB/index.php5/ITool

Conclusion

BMotion Studio 2 provides an effective way to create visualisations of formal models. The use of web technologies as the underlying graphical engine allows the user to reuse existing components found in the world wide web, like JavaScript libraries and SVG graphics. Moreover, the support of a scripting language may facilitate the creation of complex and dynamic visualisations.

The graphical representation of a formal model can be helpful for discussing the specification with non-formal method experts and for the further development of the specification, as well as indentifying inconsistencies or unexpected behaviours within the specification.

References

[1] Visualising Event-B Models with B-Motion Studio, In Proceedings of FMICS 2009, volume 5825 of Lecture Notes in Computer Science, Springer, 2009.
[2] An Approach for Creating Domain Specific Visualisations of CSP Models, In HOFM 2014, LNCS, 2014.
[3] Scalable Vector Graphics (SVG) 1.1 (Second Edition), http://www.w3.org/TR/SVG11.
[4] Cascading Style Sheets (CSS) Snapshot 2010, http://www.w3.org/TR/css-2010.
[5] Understanding Concurrent Systems, Roscoe, A.W., 2010, Springer-Verlag New York, Inc.

[6] The Theory and Practice of Concurrency, Roscoe, A. W. and Hoare, C. A. R. and Bird, Richard, 1997, Prentice Hall PTR.

Railway Interlocking Co-Simulation for Performance Analysis with Application to Stuttgart 21

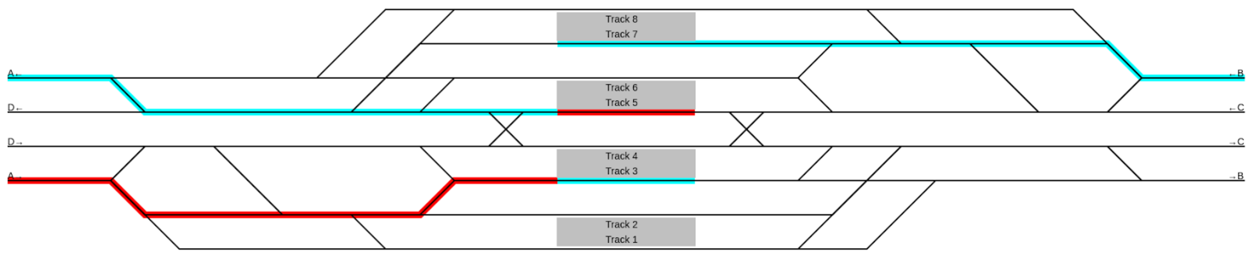
We used ProB Co-Simulation for a performance analysis of a planned railway station. The “Stuttgart21” project includes a new main station for the city of Stuttgart. Performance analysis should show if all planned train movements are possible without delays using the planned interlocking system.

We created the interlocking system with ProB. The model includes all safety critical aspects. Instead of using FMI, we used Java within a Groovy script for the co-simulation of arriving and departing trains according to timetable. The main advantages are that no license is needed and that it is more flexible than FMI. It turned out that Groovy is appropriate for this type of co-simulation. In addition, it eased the control of our graphical representation as BMotion Studio 2 is controlled by the Groovy script as well.

After running the co-simulation, the performance results are evaluated and the user gets all informations about the performance of the interlocking system with the given scenario. One could easily change some parameters of the scenario in the Groovy script and co-simulate again, without any recompiling. Compared to FMI, this is a great advantage.

We had a previous version of this model using the original ProB Plug-in with BMotion Studio 1 without co-simulation. The new model using the new ProB API and BMotion Studio 2 is about 20 times faster.

Detailed informations about co-simulation can be found in the Co-Simulation Tutorial on the ProB website.



FMI Multi-Simulation

Overview

During this period we have improved performance and stability and have added new features to the multi-simulation framework. We have also applied the tool to a new case study from WP1. The MultiSim Plug-in has also been further developed.

MultiSim Plug-in

MultiSim plug-in for Rodin is an extension that provides a generic multi-simulation capability for engineers of hybrid systems. The main purpose of this feature is to allow engineers to validate their discrete Event-B models against the realistic physical models of environment by simulating them in a single tool. The idea and design of the extension was proposed for the ADVANCE project by Michael Butler and later implemented by the team of researchers in the University of Dusseldorf and the University of Southampton. The prototype implementation was refined and updated to be based on the latest version of the Rodin platform, the Event-B EMF framework, and the ProB Java API. It was successfully validated on a number of hybrid system models, including the smart grid case study (WP2) by Critical Software.

Motivations / Decisions

Backport of the Co-Simulation Framework for ProB 1

Due to a request by WP2, we backported the framework for Co-Simulation with FMI to older ProB versions, allowing tools that are not yet using ProB Java API to run Co-Simulations.

Physical Models and FMI

One of the main goals of the ADVANCE project was to enhance the Rodin toolset specifically for the design and verification/validation of hybrid, or cyber-physical systems, through the integration of the Rodin platform with continuous-time simulation technologies. Despite the power of Event-B language, it is still restrictive when it comes to expressing the continuous aspect (primarily the environment) of hybrid systems, as that has to be abstracted to discrete terms. One of the key properties of hybrid systems is that the computational part cannot be studied separately from the environment because of their tight coupling, hence the importance of modelling the intrinsically continuous and unpredictable environment and validating Rodin models against it was the motivation to develop a solution that allows their integration and combined validation.

The concept of composing the discrete (computational) aspect of a system, modelled in Event-B, with the continuous aspect of environment, designed using complex physical modelling technologies, such as the Modelica language [1], was further developed into a multi-simulation plug-in for Rodin based on the Functional Mock-up Interface for Co-Simulation standard [2]. The FMI standard is industrially accepted, open-source, and tool-independent, which enables the support of a wide range of domains-specific modelling and simulation technologies to be integrated together. An interface for FMI 1.0 models via the JFMI library from Berkeley [3] has been developed in the ProB 2.0 package and is utilised in the multi-simulation plug-in. The plug-in itself implements a generic FMI simulation master algorithm based on a two-list evaluation concept [4] and supports continuous simulation of multiple Event-B and FMI components, therefore providing a deterministic and scalable simulation.

Visual Notation and Extensibility

In order to provide convenient composition facilities for composing Event-B and FMI components prior to simulation, the plug-in was designed as a graphical block-based editor, similar to tools such as the MATLAB Simulink and Modelica-based editors. This makes it easy for the users to import components on a canvas as blocks and connect them via input and output port signals. The editor performs an automatic validation and visual feedback of the composed graph before the start of simulation.

The plug-in is designed using standard Eclipse frameworks (EMF and GMF) with extensibility in mind. The simulation is performed on the level of a generic component, which hides its simulation semantics from the master algorithm. In other words, the plug-in can be easily extended to support additional simulation blocks besides Event-B and FMI.

Dependencies and Features

The plug-in uses the latest Event-B EMF framework, which makes it compatible with recent versions of other EMF-based Rodin tools, such as iUMLB State-Machines for modelling systems at the graphical abstraction of state charts. It also facilitates the newly introduced features of the ProB Java API for improved simulation performance. For instance, simulation of a control cycle of an Event-B component uses an LTL evaluation command to reduce the number of context switches between the ProB client and the simulation tool, which on average improves the simulation time by a factor of 8.

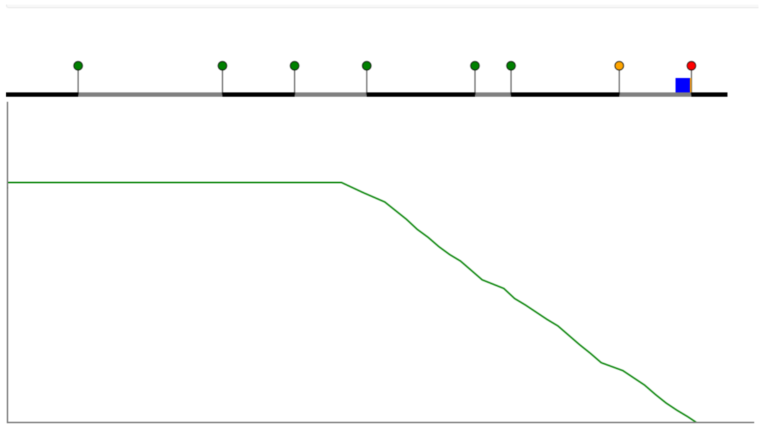
Results of the simulations are recorded both in a signal value output file and a serialised event trace file for each of the simulated Event-B components. The traces can be then reused, for instance, for visualisation and analysis using the BMotion Studio 2. The output signal values of each component can also be plotted during the simulation via the Display components, similar to the Scope blocks in MATLAB Simulink.

ETCS Co-Simulation Case Study

We used a model of an ETCS controller for a co-simulation case study. The B model includes the track structure, the positions and states of the signals, and the informations about the maximum speed allowed on a specific track section. If a train passes a balise, informations are send to the train controller which was modelled using FMI.

The train controller calculates the braking distance for the train, applying the given informations from the B model. It decides whether it is necessary to decelerate the train, if it is safe to accelerate it, or if it should keep its current velocity. The new velocity and the current train position will be returned to the B model.

We used C code for the FMI. A Groovy script was used to manage the interactions between the B model and the FMI. We created a simulation using BMotion Studio which visualises the moving of the train and its velocity. The visualisation helps understanding the model and the co-simulation.



Available Documentation

The MultiSim plug-in proposal has been presented at the 5th Rodin User and Developer Workshop [5] and the 2014 Summer Simulation Multi-Conference [6]. A validation of the tool on a landing gear case study has also been presented at the 4th International ABZ 2014 Conference [7].

Both the latest source code and the installation/usage instructions for the plug-in users are available from the git repository [8]. The instructions include a link to the FMI specification, an information on required dependencies, such as ProB 2.0, and a quick user’s guide on how to use the tool.

Conclusion

MultiSim and the ProB FMI interface provide a generic multi-simulation capability that combines formal modelling with physical simulation in order to aid the requirements and design engineers to better understand the interactions of discrete and continuous aspects of hybrid systems and to analyse system behaviour in a complex model of environment via simulation. This offers another facet of the validation of a constructed system, which should help to increase the safety and reliability assurance.

As future improvements we consider optimising the simulation performance, both in the memory footprint and execution speed, more flexible and customisable simulation semantics, in particular the mapping of Event-B models to simulation components, and possibly extending integration to other co-simulation interfaces and tools besides the FMI.

References

[1] Modelica Language, https://www.modelica.org
[2] Functional Mock-up Interface Standard, https://www.fmi-standard.org

[3] JFMI - A Java Wrapper for the Functional Mock-up Interface, http://ptolemy.eecs.berkeley.edu/java/jfmi
[4] S. Mazor, P. Langstraat. A Guide to VHDL, Second Edition. Springer, 1993.
[5] V. Savicks, M. Butler, J. Colley, J. Bendisposto. Rodin Multi-Simulation Plug-in. In, 5th Rodin User and Developer Workshop, 2014. http://eprints.soton.ac.uk/365249
[6] V. Savicks, M. Butler, J. Colley. Co-Simulating Event-B and Continuous Models via FMI. In, 2014 Summer Computer Simulation Conference. http://eprints.soton.ac.uk/365252
[7] V. Savicks, M. Butler, J. Colley. Co-simulation Environment for Rodin: Landing Gear Case Study. In, 4th International ABZ 2014 Conference. http://eprints.soton.ac.uk/364883
[8] MultiSim Git repository: https://github.com/snursmumrik/rms2/tree/multisim

ProB Constraint Solving Kernel

Overview

During the lifetime of the ADVANCE project major performance improvements have been achieved in the constraint solving kernel of ProB. This has continued in the last period of ADVANCE. Below are three constraint solving benchmarks, and the solution times of ProB just before the beginning of the ADVANCE project and now. The third example (peacable armies of queens) is a benchmark in the constraint programming community (see [1]). More detailed results can be seen in some of the articles below.

Benchmark	ProB 1.3.4-b16 (Sep'11)	ProB 1.3.7-b6 (25 Aug'13)	ProB 1.5.0-b3 (Oct'14)	Speedup (Sep'11-Oct'14)
NQueens 50	1.07 sec	0.33 sec	0.38 sec	2.8
Sudoku Hex	6.72 sec	0.62 sec	0.59 sec	11.4
PeacableQueens 8*8	> 1000 sec	> 1000 sec	0.61 sec	>1639

A lot of effort in the research and development went into the detection of enumeration warnings. This is now robust and has enabled the use of ProB not only as a Disprover but also as a prover (see WP3). The detection and treatment of recursive functions has also been improved. These features are also crucial for automatically detecting potentially infinite sets and thus for dealing automatically with many Event-B Theories.
A secondary, independently developed tool PyB (https://github.com/hhu-stups/pyB/) was developed to double check the results of the ProB constraint solver for high-assurance applications. PyB is an B-interpreter written in python. It can double check results of ProB by

using computed variable and constant values and reevaluating invariant and property-clauses. It may be used for data validation and was tested with industry-size machines.

Motivations / Decisions

- The constraint solver is at the heart of ProB, and its capabilities and performance is crucial for many applications. In particular, the constraint-based model-based testing is highly reliant on the capabilities and performance of the solver.
- ProB moved to SICStus 4.3 which includes a JIT compiler and enables generating a 64-bit version of ProB on Windows
- We provided an improved REPL (Read-Eval-Print-Loop) as requested by WP1 to allow the introduction of temporary values using a let statement in order to help debug B models and their states
- We enlarged the ability of ProB to deal symbolically with operators (set difference, set intersection, …), supported the Event-B cond operator for recursive function construction (relevant for Theory Support), and added many improvements to the constraint solving kernel such as the reification of set-cardinality
- We developed a secondary toolchain PyB and strengthened the validation of the core ProB kernel.

Available Documentation

- [1] Michael Leuschel, David Schneider. Towards B as a High-Level Constraint Modelling Language ABZ 2014 http://link.springer.com/chapter/10.1007%2F978-3-662-43652-3_8
- [2] Jens Bendisposto, Sebastian Krings, Michael Leuschel. Who watches the watchers: Validating the ProB Validation Too. In Proceedings of the 1st Workshop on Formal-IDE, EPTCS 149, 2014, Electronic Proceedings in Theoretical Computer Science, 2014. http://www.stups.uni-duesseldorf.de/w/Special:Publication/fide14toolchain
- [3] John Witulski, Michael Leuschel. Checking Computations of Formal Method Tools - A Secondary Toolchain for ProB. In Proceedings of the 1st Workshop on Formal-IDE, EPTCS 149, 2014, Electronic Proceedings in Theoretical Computer Science, 2014. http://www.stups.uni-duesseldorf.de/w/Special:Publication/fide14pyb
- Daniel Plagge, Michael Leuschel. A Practical Approach for Validation with Rodin Theories. Rodin Workshop 2014. Toulouse. June 2014. http://wiki.event-b.org/index.php/Rodin_Workshop_2014.
- Standard benchmark example illustrating the capabilities of the ProB kernel: http://www.stups.uni-duesseldorf.de/ProB/index.php5/Peaceable_Armies_of_Queens
- A small tutorial showing constraint solving capabilities in REPL: http://www.stups.uni-duesseldorf.de/ProB/index.php5/Sudoku_Solved_in_the_ProB_REPL
- ProB Logic Calculator (with many examples: http://www.stups.uni-duesseldorf.de/ProB/index.php5/ProB_Logic_Calculator
- Tutorial on constraint solver (needs updating): http://www.stups.uni-duesseldorf.de/ProB/index.php5/Tutorial_Understanding_ProB%27s_Constraint_Solver
- Invited Talk at VPT (CAV Workshop) at Vienna Summer of Logic http://refal.botik.ru/vpt/vpt2014/program.html

- ProB has been used “out-of-the-box” for Rodin theories by Thales for their interlocking model, building ProB BMotionStudio visualizations on top (see slides of the Advance Industry Day http://www.advance-ict.eu/sites/www.advance-ict.eu/files/Thales-Duesseldorf.pdf)

Conclusion

Many new capabilities and performance improvements were integrated into the ProB solver over the last period. ProB is, to our knowledge, the only solver capable of dealing with **large** integers and data values, automatically detecting potentially **infinite** sets, and attempting to solve constraints over **higher-order** data structures.

Test case generation

In model-based testing you want to generate traces and values for your model which satisfy a certain coverage criterion. There are two ways this coverage can be achieved systematically: using model checking or using a constraint-based approach. Below we describe progress on each of those approaches, as well as the implementation of a new coverage criterion (MC/DC).

Model Checking Based Test Case Generation

Overview

We have successfully applied the model checking based test-case generation to WP5 examples as well as to an important external case study. This case study profited from improvements made to the ProB toolset for model based testing. In particular, a way to guide the model checking using a predicate of interest was introduced (command line option `-scope PRED`) and a convenient way to extract all non-looping paths (aka test cases) was added (using the command line option `-all_paths`). This made it possible to generate test cases much faster than the previous constraint-based approach.

Motivations / Decisions

As part of the Dstl (Defence Science and Technology Laboratory) project HASTE, the University of Southampton conducted a case study based upon the FADEC (Full Authority Digital Engine Control) system for a hypothetical helicopter.

After some preliminary analysis, a subpart of the FADEC system was chosen for detailed modelling. This subsystem consisted of an Anti-Ice Bleed Valve (AIBV) and its Controller. STPA hazard analysis was performed on the AIBV, and a detailed Event-B model was constructed that captured the possible failure modes of the valve actuator and valve position sensor.

During the development of the model, ProB found a counterexample to an important safety requirement - that the controller should have an accurate model of the valve position. Specifically, ProB identified that actuator and sensor failures could be correlated, and that this could result in the masking of these failures from the controller. The safety requirement was consequently changed to reflect that in these circumstances it is not possible to guarantee the correctness of the controller's process model. Thus the modelling process identified a requirement on the physical valve assembly (that actuator and sensor failures must be uncorrelated), and the corresponding safety requirement became provable under the assumption that this physical requirement had been met.

Test cases were generated from the resulting Event-B model by first developing a test plan based upon bounding the number of control loops, and partitioning the input temperature (from which the controller determines the required valve position) into equivalence classes, and then refining the Event-B model to apply this test plan. This was done by adding ghost variables to the model that record the number of control loops performed and which equivalence classes have been covered. The resulting model was then passed to ProB to generate the test cases.

In our case study, bounding the number of control loops to 3, and partitioning the input temperature into 6 equivalence classes (in order to plot the control law of the controller), ProB generated 12,648 test cases in about 15 minutes. These test cases cover all possible sequences of actuator and sensor failures of the AIBV identified during the STPA hazard analysis.

Available Documentation

http://www.stups.uni-duesseldorf.de/ProB/index.php5/Tutorial_Model-Based_Testing#Model-Checking_Based_Testcase_Generation

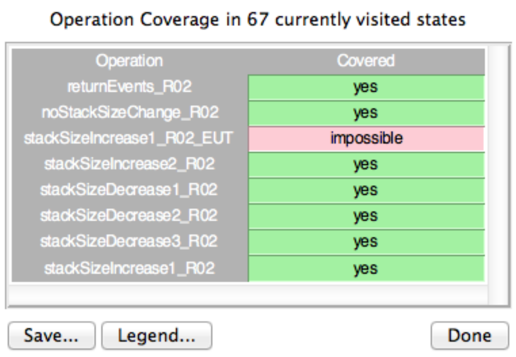
Conclusion

We have added some new features to ProB to ease model checking based test case generation. This has considerably reduced the runtimes in some cases and already enabled one important application outside of the Advance project.

Constraint-Based Test Case Generation

Overview

Within the project we have also advance the state of the model-based testing technology for Event-B. The performance of the constraint-based test case generation has been considerably improved (see some numbers in the model-based testing section below), and the algorithm has been made more intelligent by using statically computed enabling and feasibility information. This can both considerably speed up the test-case generation algorithm (factors from 2 to 10 were typical in our experiments) and provide better user feedback. Indeed, often the algorithm can terminate earlier, as the algorithm is not trying to cover infeasible events, and then provide the user with informative feedback that certain events can definitely never be covered.


Operation Coverage in 67 currently visited states

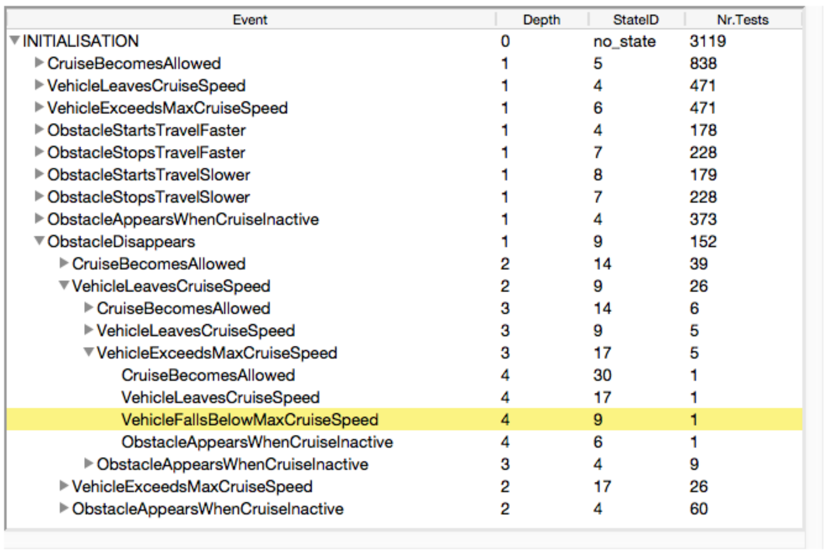
Operation	Covered
returnEvents_R02	yes
noStackSizeChange_R02	yes
stackSizeIncrease1_R02_EUT	impossible
stackSizeIncrease2_R02	yes
stackSizeDecrease1_R02	yes
stackSizeDecrease2_R02	yes
stackSizeDecrease3_R02	yes
stackSizeIncrease1_R02	yes

Save... Legend... Done

This table highlights the performance improvement achieved during the last 18 months of the project, on a particular benchmark case study. For the smart card case study described below, the improvements were even more marked.

Benchmark	ProB 1.3.6-rc1 (Feb'13)	ProB 1.5.0-b2 (Oct'14)	ProB 1.5.0-b4 (Nov'14)
Adaptive Cruise Controller (4)	14.45 sec	3.07 sec	2.50 sec
Adaptive Cruise Controller (5)	137.06 sec	13.56 sec	8.02 sec

Recently we have also added an inspector for the test-cases. Here is the overview of the 3119 test cases generated for complete coverage (max. search depth 5) of the adaptive cruise controller model from the table above.


Event	Depth	StateID	Nr.Tests
▼INITIALISATION	0	no_state	3119
▶CruiseBecomesAllowed	1	5	838
▶VehicleLeavesCruiseSpeed	1	4	471
▶VehicleExceedsMaxCruiseSpeed	1	6	471
▶ObstacleStartsTravelFaster	1	4	178
▶ObstacleStopsTravelFaster	1	7	228
▶ObstacleStartsTravelSlower	1	8	179
▶ObstacleStopsTravelSlower	1	7	228
▶ObstacleAppearsWhenCruiseInactive	1	4	373
▼ObstacleDisappears	1	9	152
▶CruiseBecomesAllowed	2	14	39
▼VehicleLeavesCruiseSpeed	2	9	26
▶CruiseBecomesAllowed	3	14	6
▶VehicleLeavesCruiseSpeed	3	9	5
▼VehicleExceedsMaxCruiseSpeed	3	17	5
CruiseBecomesAllowed	4	30	1
VehicleLeavesCruiseSpeed	4	17	1
VehicleFallsBelowMaxCruiseSpeed	4	9	1
ObstacleAppearsWhenCruiseInactive	4	6	1
▶ObstacleAppearsWhenCruiseInactive	3	4	9
▶VehicleExceedsMaxCruiseSpeed	2	17	26
▶ObstacleAppearsWhenCruiseInactive	2	4	60

Smart Card Case Study

The improved test case generation algorithm has been applied in the context of an external smart card case study. Everybody use smart cards. For the users, these cards represent the key to accessing a service. For providers, smart cards are doors to bigger systems, such as banks, and they must ensure that an attacker cannot compromise the entire system. That is why the verification process must ensure their resistance against attacks. Formal methods have already been used for a while to verify these systems, but some components are really hard to verify and even if we have a verified model, we can only verify what has been modeled.

In our research, we want to verify our system with blackbox penetration testing. We used the Event-B language to model our SUT (System Under Test). We make the assumption that the accepted behavior of our SUT is completely represented by the initial model, so the unmodeled behavior must represent unacceptable behavior for our SUT. Using specification mutation, we can generate an model corresponding to a subset of the unmodeled behavior. To performed this mutation, we have developed a Rodin plug-in. To extract tests from the mutants, we use a ProB model-based testing algorithm. These tests are then concrete and are sent to our SUT. With this technique, we have found several weaknesses in the embedded security systems of smart cards. During the last months of the project we managed to reduce the runtime for achieving complete mutant coverage from two years computation time down to less than two hours, mainly by improving the constraint-based test case generation algorithm and using the ProB Java API for constructing mutants. Indeed, using the original approach (creating one Event-B machine per mutant) and the original constraint-based test case generation algorithm the time to compute all test cases was estimated at around two years. Using a new approach (putting all mutants in one Event-B machine) and using the improved constraint-based test case generation algorithm, this time has been reduced to less than 1 hour 20 minutes. In that time, 820699 paths were checked by the ProB constraint solver, resulting in 275,301 feasible paths and test cases for 221 reachable mutants and determining that the remaining 700 mutants were unreachable within the model. The results will be published in an academic article.

Motivations / Decisions

- Constraint-Based test case generation is important for models involving real-data, resulting in a large number of initialisation and/or event instances. However, due to repeated calls to the constraint-solver, the performance is often a major issue. Hence we looked at various ways to improve the existing algorithm's performance.

Available Documentation

- http://www.stups.uni-duesseldorf.de/ProB/index.php5/Tutorial_Model-Based_Testing#Model-Checking_Based_Testcase_Generation
- Aymerick Savary, Marc Frappier, Jean-Louis Lanet:
- Detecting Vulnerabilities in Java-Card Bytecode Verifiers Using Model-Based Testing. IFM 2013: 223-237. http://link.springer.com/chapter/10.1007%2F978-3-642-38613-8_16
- Aymerick Savary, Marc Frappier, Jean-Louis Lanet. Toolbox for penetration testing based on Rodin and ProB. Rodin Workshop 2014. Toulouse. June 2014. http://wiki.event-b.org/index.php/Rodin_Workshop_2014

Conclusion

By using enabling analysis and feasibility computed beforehand by the constraint solver, we have improved the performance of the constraint-based test case generation algorithm. In an external application for SmartCards we have reduced the overall runtime considerably, from two years to less than two hours.

MC-DC Coverage

Overview

MC/DC (Modified Condition/Decision Coverage) is a coverage criterion which is used in avionics. One core idea is ensure that for every condition in our code or model we generate at least one test where the condition independently plays an observable role in the outcome of a decision. In other words, if the condition were to be true rather than false (or vice versa) the decision would be changed. In still other words, we are trying to find an execution context where the result of the particular condition under test is crucial for the observable outcome.

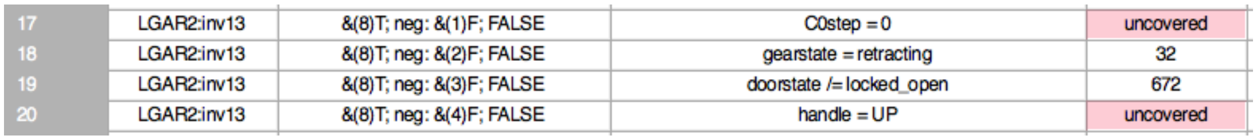
In the context of Advance and Event-B, one question is what constitutes a **condition** and what constitutes a **decision**. In WP4 we view a **condition** to be an atomic predicate (e.g., x>y) not involving logical connectives (&, or, not, =>, □). There is, however, the possibility of treating a bigger predicate as atomic (e.g., an entire guard of an event). The reason for this is to provide better feedback to the user and to reduce the number of coverage criteria if desired. At the moment, we also consider a universally or existentially quantified formula to be a condition.
A **decision** constitutes the fact whether an event is enabled or not. However, we also enable MC/DC to be applied in other contexts, e.g., to analyse invariants; in that setting a decision is whether an invariant is true or false.

We have implemented a MC/DC coverage analysis which examines all visited states and computes the MC/DC coverage for a series of predicates under consideration. Currently this predicate is called in two ways:

- for the coverage for the invariants, where the invariant is expected to be true (i.e., the analysis does not expect to find states where the invariant is false)
- for the coverage for the event guards. Here it is expected that every event can be enabled or disabled.

The algorithm is parametrised by an optional depth bound: if the depth bound is reached, then all sub-predicates are considered to be atomic decisions.
The analysis can be used to measure the coverage of tests, but can also be used after an exhaustive model check to detect vacuous guards and vacuous invariants.
The MC/DC analysis has been integrated into ProB and has been successfully applied to various case studies (e.g., the landing gear case study in WP5).
Below is a screenshot of part of the output for an early version of the WP5 landing gear case study. It shows that parts of the invariant inv13 `not(C0step = 0 & (gearstate = retracting & doorstate /= locked_open & handle = UP))` cannot be set independently

to false. This highlights a redundancy in the invariant; indeed the simpler to understand (but stronger) invariant `not(gearstate = retracting & doorstate /= locked_open)` holds.



17	LGAR2:inv13	&(8)T; neg: &(1)F; FALSE	C0step = 0	uncovered
18	LGAR2:inv13	&(8)T; neg: &(2)F; FALSE	gearstate = retracting	32
19	LGAR2:inv13	&(8)T; neg: &(3)F; FALSE	doorstate /= locked_open	672
20	LGAR2:inv13	&(8)T; neg: &(4)F; FALSE	handle = UP	uncovered

Motivations / Decisions

- Meeting industrial needs for certification, in particular in light of DO-178.
- Helping find vacuous parts and improve and clarify the model

Available Documentation

- see deliverable D5.3 from workpackage WP5, in particular the Landing Gear Tutorial
- http://www.stups.uni-duesseldorf.de/ProB/index.php5/Tutorial_Model-Based_Testing#Model-Checking_Based_Testcase_Generation

Conclusion

We have successfully implemented and applied MC/DC coverage for Event-B and classical B models. This feature has proven to be surprisingly useful for model validation.

Code Generation.

Overview.

In the previous project, DEPLOY, work on code generation had begun but was still at the stage of a very early prototype. In ADVANCE, progress was made on several fronts. Firstly, the existing tool was upgraded to improve usability. The various editing tasks were properly integrated, and various aspects were automated. The tool was also extended with new features, such as the ability to generate code from state-machines [1]. The use of the theory plug-in for adding type extensibility to code generation was reported in [2]. For the co-simulation effort, a feature for generating code from component diagrams was added [5]. C code satisfying the FMI standard, for use in FMUs, is generated. A simple tool is also used to package compiled code into an FMU, for use in co-simulation. Various research tracks were followed, including instrumenting code to vary rates at which certain paths in a simulation are taken [1]; the use of templates for code re-use and code configuration, using code-injection[4]; and exploratory work on translation to more complex data types using the theory plug-in [5].

A C-code generator for FMI.

In ADVANCE, new tools have been developed which feature co-simulation with Event-B and non-Event-B models. The existing Event-B code generators supported generation of Ada [3], and to a limited degree, OpenMP C, and Java. Work undertaken in ADVANCE adds additional functionality to generate C code from Event-B components with state-machines [1]. Some components describe the behaviour of embedded controllers. Code generated from the controller components can be used to replace Event-B models in the simulation. The code conforms to the FMI standard, and can be packaged as an FMU, with or without compiled binaries. It is useful to automatically generate controller code to test it against continuous models of the controller environment. The code can also be used for the deployed product [5].

Code from State-machine Diagrams.

The work on generating code from state-machine diagrams [1] consists of two parts. The first adds the capability to generate code from Event-B state-machine diagrams. The second uses the code to simulate concurrently executing state-machines in a single task. We can instrument the code to guide the simulation by controlling the relative rate that non-deterministic transitions are traversed in the simulation. The first part of this work formed the basis of the FMI-C code generator.

Templates in Code Generation.

We can configure the code generator for deployment on different target systems. Templates [4] can be used to avoid hard-coding 'boilerplate' code and can be used to merge with code generated from the formal model. We developed a lightweight approach where tags (i.e. tagged mark-up) can be placed in source templates. The template-processors we introduce may be of use to other plug-in developers wishing to merge a 'source' text file with some generated output. The template driven approach was used when developing the FMI code generator.

Java Code Generation Improvements.

As part of the maintenance activity, the tool has been updated with improved translators for generating Java code; this includes integrating Event-B projects and processing the generated Java using Java Natures and Builders. We have provided usability enhancements including automatic flattening of invariants and events; and automatic inference of typing annotations and parameter directions. This should result in a reduced workload for the developer, and requires less specialized knowledge.

A Theory for implementable Sets and Functions.

Code generation involving abstract Event-B sets and functions has largely been avoided in our work until now. In ADVANCE, we reported on the integration of code generation and the theory plug-in [2]. We advanced the capability of the tool by creating Theories for implementable sets and functions [5], using polymorphic type parameters to describe implementations involving generic (parametrised) sets and maps. For sets, we introduce operators for typing, and initialisation. We support the usual set operations, and require some new operators to facilitate code generation. A translation to Java is defined in the theory, which makes use of the new operators. This is supported by a Java implementation of a set, which is intended to be a refinement of the model. Functions are similar in that we introduce typing, and initialisation, but with update and lookup operators. The translation to Java is backed by an implementation using a HashMap to store the domain and range values as key value pairs.

Papers:

[1] Edmunds, A. and Colley, J. and Butler, M. *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. http://deploy-eprints.ecs.soton.ac.uk/455/2/DS_EventB_2012.pdf

[2] Edmunds, A. and Butler, M. and Maamria, I. and Silva, R. and Lovell, C. *Event-B code generation: type extension with theories.* In: ABZ 2012, 19-21 June 2012, Pisa, Italy. http://eprints.soton.ac.uk/336226/1/ABZ2012_short_v20120202.pdf

[3] Edmunds, A. and Rezazadeh, A. and Butler, M. *Formal modelling for Ada implementations: Tasking Event-B.* In: Ada-Europe 2012: 17th International Conference on Reliable Software Technologies, Stockholm. http://deploy-eprints.ecs.soton.ac.uk/375/1/AdaEurope2012.pdf

[4] Edmunds, A. *Templates for Event-B Code Generation.* In Proceedings of 4th International ABZ Conference (ABZ 2014). http://eprints.soton.ac.uk/364265/1/Templates4CG.pdf

[5] Edmunds, A. *Code Generation – Tool Developments (2014).* In 5th Rodin User and Developer Workshop. http://eprints.soton.ac.uk/370683/1/RUDW2014.pdf

[6] A. Salehi Fathabadi, , C. Snook, and M. Butler (2014) Applying an integrated modelling process to run-time management of many-core systems. In, 11th International Conference on Integrated Formal Methods (iFM), Sep 2014. http://eprints.soton.ac.uk/366747/

Conclusion.

We performed initial experiments with the code generator on a simplified version of the WP2 smart gird case study. While the initial results are promising, more detailed refinement and decomposition of the WP2 models would be required to be able to generate efficient implementation code using the code generator. The current WP2 models are at a high abstraction level because the primary goal of WP2 was to address specification and verification of system-level properties and on model validation through co-simulation. The code generator is intended for generation of refined, close-to-implementation, Event-B models. On a separate UK EPSRC-funded project on many core systems, the code generator has been used to generate part of a runtime management controller for energy reduction in a multi-core system [6].

We believe that we have advanced the code generation capabilities of Event-B during ADVANCE such that it is beginning to be a factor in the appeal of the tool to potential industrial users. Work is being undertaken, in collaboration with industry, to further understand how the technology can be leveraged, towards the end of a development iteration. In addition, more needs to be done to understand the role of code generation in co-simulation environments and model-based test generation.