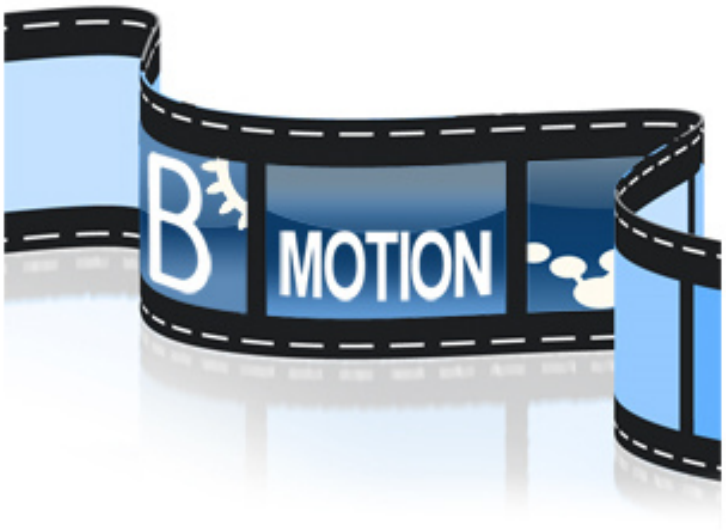

BMotionWeb
Handbook (v0.1.0)

ProB Edition

Lukas Ladenberger (Editor)

Advance This work is sponsored by the ADVANCE Project

Contents

Chapter 1

Introduction

1.1 Overview

This handbook consists of five parts:

Introduction (Chapter 1) You are reading the introduction right now. Its purpose is to help you orient yourself and to find information quickly.

First Steps (Chapter 2) If you are completely new to BMotionWeb, this section is a good way to get up to speed quickly. It guides you through the installation and usage of the tool.

BMotionWeb for Event-B and Classical-B (Chapter 4) This section provides a documentation of BMotionWeb for creating visualizations of Event-B or Classical-B models.

Frequently Asked Questions (Chapter 5) Common issues are listed by category in the FAQ.

Index We included an index particularly for the print version of the handbook, but it may be useful in the electronic versions as well.

1.1.1 Formats of this Handbook

The handbook comes in various formats:

Online You can access the handbook online.

PDF Both online versions also include a link to the PDF version of the handbook.

1.2 Conventions

We use the following conventions in this manual:

 Checklists and milestones are designated with a tick. Here we summarize what we want to learn or should have learned so far.

 Useful information and tricks are designated by the information sign.

 Potential problems and warnings are designated by a warning sign.

 Examples and Code are designated by a pencil.

We use `typewriter` font for file names and directories.

We use sans serif font for GUI elements like menus and buttons. Menu actions are depicted by a chain of elements, separated by "⟩", e.g. File ⟩ Open Visualization.

1.3 ADVANCE

This work has been sponsored by the Advance project[1]. ADVANCE is an FP7 Information and Communication Technologies Project funded by the European Commission. The overall objective of ADVANCE is the development of a unified tool-based framework for automated formal verification and simulation-based validation of cyber-physical systems.

The ADVANCE project is unique in addressing both simulation and formal verification within a single design framework.

Unification is being achieved through the use of a common formal modelling language supported by methods and tools for simulation and formal verification. An integrated tool environment is providing support for construction, verification and simulation of models.

ADVANCE is building on an existing formal modelling language - Event-B - and its associated tools environment - Rodin - with strong support for formal verification. In ADVANCE, Rodin is being further strengthened and augmented with novel approaches to multi-simulation and testing.

1.4 Creative Commons Legal Code

The work presented here is the result of an collaborative effort that took many years. To ensure that access to this work stays free and to avoid any legal ambiguities, we decided to formally license it under the Creative Commons NonCommercial ShareAlike License.

[1] http://www.advance-ict.eu/

Chapter 2

First Steps

2.1 Installation and Start

Start off by downloading BMotionWeb for your operating system. You can find the latest version of the tool at http://www.stups.hhu.de/ProB/index.php5/BMotionWeb. Decompress the archive and expand the directory if necessary. Navigate to the application folder and start BMotionWeb by executing the `bmotion-prob` binary. After a short loading time you should see the window shown in Figure 2.1.

2.2 Open a Visualization

To open a visualization, click on the box in the middle of the window and select the BMotionWeb manifest file (see 3.1.1) of the visualization or just drag and drop the BMotionWeb manifest file into the box. You can also open a visualization via the top menu: File ⟩ Open Visualization.

2.3 Create a new Visualization

To create a new visualization choose File ⟩ New Visualization. A window will be opened asking you for some additional information (e.g. the id and name of your visualization). Enter your data and press on OK. Now, the tool asks you for a location (a folder) where you want to save your visualization. In the next step, the tool asks you for a model to be visualized. Select a model and click on OK. This will start the fresh visualization. The tool will create a bunch of files into the selected folder:

`bmotion.json`: The `bmotion.json` file is the root file of your BMotionWeb visualization (also called BMotionWeb manifest file). It contains the configuration formatted using JSON (JavaScript Object Notation)[1].

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

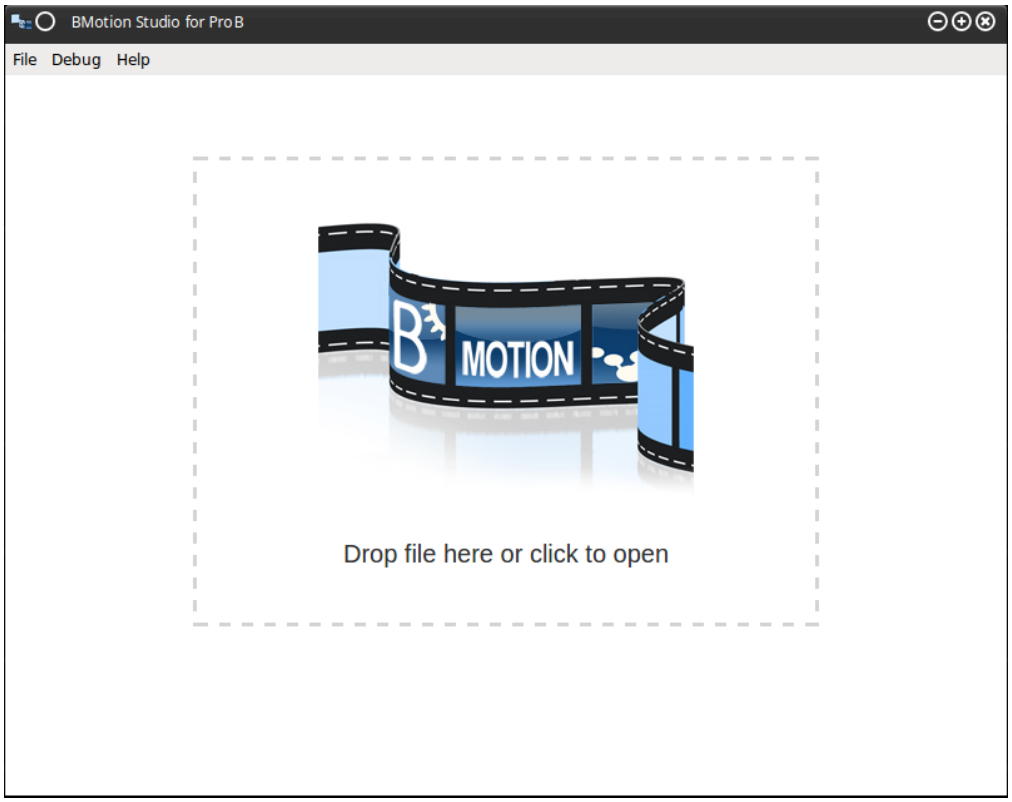

Figure 2.1: BMotionWeb Desktop Application

Section 3.1.1 contains a full list of available options.

`script.js`: In the JavaScript file you can setup observers and actions (see Section 4.2). Moreover, the user can take advantage of the entire JavaScript language. There exist are a lot of libraries for JavaScript that you can apply to create custom visualizations. For instance, it exists libraries for generating chart and plot diagrams.

`index.html`: The HTML file contains the reference to the `scripts.js` file and to the `visualization.svg` file.

`visualization.svg`: The actual SVG visualization. The user is not restricted to an editor in order to create a visualization. The user can make use of the integrated visual editor or any other tool that supports the creation of SVG graphics.

`bms.api.js`: JavaScript library that is needed for running the visualization. Please do not edit this file!

Chapter 3

BMotionWeb

3.1 Visualization Template

At the heart of an interactive formal prototype, one finds a *visualization template*. It is the part of the interactive formal prototype that is developed by the user. It describes the visualization and the gluing code (observers and interactive handlers) for the interactive formal prototype. An important design decision was to make the full web-technology stack available to the developer for creating a visualization template. The benefit of this design decision is that the visualization template becomes flexible since external resources, such as SVG images and third party JavaScript libraries, can be reused. Indeed, this can save time for developing an interactive formal prototype and may provide a large selection of reusable SVG images and JavaScript libraries. This design decision can also help the developer to create complex interactive formal prototypes, e.g. with numerous or repeated graphical elements. In general, a visualization template consists of several files which are described in the following subsections.

3.1.1 Manifest File

A visualization template is identified by a *manifest file*. The manifest file is the root file of every interactive formal prototype. It contains the configuration for the interactive formal prototype in JSON (JavaScript Object Notation) format.[1] Table 3.1 gives an overview of the available options. The table shows the option's name, its type, a short description, and denotes if the option is required or optional. Listing 3.1 exemplifies the use of a manifest file based on the interactive formal prototype of the Event-B simple lift system.

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

Table 3.1: Available options for BMotionWeb manifest file

Name	**Type**	**Required**	**Description**
id	string	yes	Unique id of the interactive formal prototype.
name	string	no	The name of the interactive formal prototype.
template	string	yes	The relative path to the HTML template file (e.g. "template.html").
groovy	string	yes	The relative path to the groovy script file (e.g. "script.groovy").
model	string	yes	The relative path to the formal specification file that should be animated (e.g. "model/mymodel.mch").
modelOptions	map	no	A key/value map defining the options for loading the model - The available options are dependent on the animator and formalism.
autoOpen	array	no	The user can specify the ProB views which should be opened automatically when running the interactive formal prototype - The following views are available for ProB animations (Event-B, classical-B and CSPM): *CurrentTrace*, *Events*, *StateInspector* and *ModelCheckingUI*.
views	list	no	List of additional views - A view object has the following options:
id	string	yes	Unique id of the view.
name	string	no	The name of the view.
template	string	yes	The relative path to the HTML template file of the view (e.g. "view1.html").
width	numeric	no	The width of the view.
height	numeric	no	The height of the view.

```
1 {
2   "id": "lift",
3   "name": "Simple lift system",
4   "template": "lift.html",
5   "groovy": "script.groovy",
6   "model": "model/m2.bcm",
7   "autoOpen": [
8     "CurrentTrace",
9     "Events"
10  ]
11 }
```

Listing 3.1: Example manifest file for the simple lift system (JSON)

3.1.2 Visualization Files

The HTML template file that is linked in the manifest (see line 4 in Listing 3.1) is the starting point for developing the actual visualization for the interactive formal prototype. The snippet in Listing 3.2 shows an example HTML template file for the simple lift system.

```
1 <html>
2 <head>
3   <title>Simple lift system visualization</title>
4 </head>
5 <body>
6   <script src="bms.api.js"></script>
7   <script src="lift.js"></script>
8   <div bms-svg="lift.svg"></div>
9 </body>
10 </html>
```

Listing 3.2: HTML template file for simple lift system (HTML)

In general, a visualization in BMotionWeb makes use of SVG. For this, BMotionWeb provides a special attribute called *bms-svg* that takes a relative path to an SVG image file as its value (see line 8). The attribute renders the entered SVG image file within the visualization and registers it in the visualization template. A registered SVG image file can be edited by means of the built-in visual editor in BMotionWeb which is described in Section 4.4. Since the SVG image file is an external file it can also be edited with any other SVG editor. As an example, consider Fig. 3.1. The left side of the figure shows the SVG image file for the simple lift system and the right side demonstrates the SVG file rendered in the interactive formal prototype. In addition to SVG, BMotionWeb also makes the full web-technology stack available to the user in order to create a visualization (i.e. the user can apply other web-techniques with HTML5, CSS and JavaScript).

In line 6 we reference the JavaScript file *bms.api.js*. This provides the BMotionWeb JavaScript API with functions, e.g. to register observers and interactive handlers. The developer can make use of this API by referencing an additional JavaScript file which contains

custom JavaScript code (see *lift.js* in line 7). The BMotionWeb JavaScript API is described in Section 4.2.

3.1.3 Groovy Script File

The developer can optionally define a Groovy[2] script file (see line 5 in Listing 3.1) to link custom Groovy or Java code to the interactive formal prototype that is evaluated on the server side. Within the Groovy script file the developer can also make use of the BMotionWeb Groovy API with functions, e.g. to control the integrated animation engine or to register external methods that can be triggered from the client side (JavaScript). As an example, using the Groovy API the developer may query an external database or make some complex computations based on the information coming from the animated formal specification (e.g. state information). The Groovy API is described in Section 4.3 in more detail.

3.2 Working with Graphical Elements

The web-technologies used for developing a visualization provide us with several predefined graphical elements and techniques for creating and styling them. For instance, HTML provides elements like tables, buttons and lists, and SVG provides elements like shapes and images. These elements are also referred to *DOM* elements.[3] With CSS we have a comprehensive technique to define the style and layout for HTML and SVG elements. BMotionWeb uses these techniques as the basis for implementing the graphical element concept introduced in **??** and for linking them to observers and interactive handlers.

In order to identify and to manipulate a graphical element within a visualization, BMotionWeb uses jQuery.[4] jQuery is a JavaScript library for *selecting* and *manipulating* DOM elements in an HTML document. It extends the CSS selector syntax[5] to provide selectors based on the id, name, classes, types, attributes and many more properties of a DOM element. As an example, consider the JavaScript snippet in Listing 3.3. It shows two examples for selecting and manipulating graphical elements based on the SVG image shown in Fig. 3.1. With jQuery we can select the graphical element that represents the lift door by its id as demonstrated in line 1 (the prefix "#" is used to match a graphical element by its id). Once an element is selected, jQuery provides us with a reference to the selected element and allows us to manipulate it, e.g. by changing its attributes. For instance, in line 2 we set the *fill* attribute of the door to the color *gray*. We can also select and manipulate multiple graphical elements as demonstrated in lines 4 and 5, where we select all *ellipse* graphical elements which have a *data-floor* attribute (line 4) and color them all green (line 5). For a comprehensive list of jQuery selectors we refer the reader to the jQuery selectors API documentation.[6]

[2] http://groovy-lang.org.
[3] http://www.w3schools.com/js/js_htmldom.asp.
[4] https://jquery.com.
[5] https://www.w3.org/TR/css3-selectors.
[6] http://api.jquery.com/category/selectors.

```
var door = $("#door");
door.attr("fill", "gray");

var allRequestButtons = $("ellipse[data-floor]");
allRequestButtons.attr("fill", "green");
```

Listing 3.3: Example for selecting and manipulating elements using jQuery (JS)

```
<svg width="220" height="340"
  xmlns="http://www.w3.org/2000/svg">
 <g id="lift_system">
  <g id="lift">
   <rect fill="white" stroke="black"
     height="330" width="100" y="5" x="50"/>
   <rect id="door" fill="gray" stroke="black"
     height="80" width="70" y="245" x="65" />
   <text fill="black" y="58" x="165">Floor 1</text>
   <text fill="black" y="182" x="165">Floor 0</text>
   <text fill="black" y="290" x="165">Floor -1</text>
  </g>
  <g id="request_buttons">
   <ellipse id="bt_1" data-floor="1"
     ry="11" rx="11" cy="54" cx="22" fill="gray"/>
   <ellipse id="bt_0" data-floor="0"
     ry="11" rx="11" cy="177" cx="22" fill="gray"/>
   <ellipse id="bt_-1" data-floor="-1"
     ry="11" rx="11" cy="285" cx="22" fill="gray"/>
  </g>
 </g>
</svg>
```

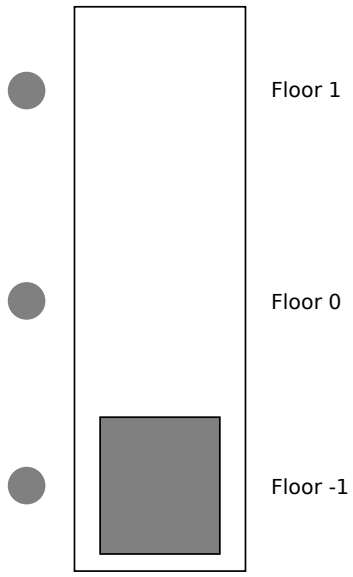

Figure 3.1: SVG image of simple lift system: source (left) and rendered (right)

Chapter 4

BMotionWeb for Event-B and Classical-B

4.1 Tutorial

The objective of this chapter is to get you to a stage where you can use BMotionWeb to visualize Event-B or Classical-B models. We expect that you have already downloaded the BMotionWeb tool (see Section 2.1).

You should be able to work through the tutorial with no or little outside help. We encourage you not to download solutions to the examples but instead to actively build them up yourself as the tutorial progresses.

4.1.1 Preparation

Let's start by creating a new visualization template as described in Section 2.3.

4.1.2 The Formal Model

We are going to create a visualization for a simple lift system that allows movement of a single lift cage between three floors. The door of the lift can be closed and opened - all in response to the pressing of floor call and cage send buttons.

You can download the Event-B model here[1]. Decompress the archive and put the files into a new folder called `model` relative to your `index.html` file.

4.1.3 Link the Model with the Visualization

The first step consists of linking the model with the visualization. For this, open the BMotionWeb manifest file with an editor of your choice and set the *model* path property to "model/MLift.bcm". This links the visualization with the Event-B machine called "MLift".

[1]The URL of the resource is: http://nightly.cobra.cs.uni-duesseldorf.de/bmotion/bmotion-prob-handbook/nightly/files/EventBLift.zip

Linking a model within the BMotionWeb manifest file will automatically load the model, when starting the visualization (see Section 4.1.5).

4.1.4 Create the Actual Visualization

Please download the prepared lift.svg[2] file and open it with Inkscape as demonstrated in Figure **??**. Feel free to modify and explore the SVG graphic. In order to link graphical elements of the SVG graphic with the formal model later, we have to give them identifiers. For this, select an element in Inkscape, open the context menu and select Object Properties. A popup window should be opened as demonstrated in Figure **??**. As an example, we give the graphical element that represents the door (the gray filled rectangle), the id "door". In Section 4.1.6 we explain how we can use this information in order to establish *observers*. If you are satisfied with your SVG graphic, save it as a plain SVG graphic with File ⟩ Save As. Select Plain SVG (*.svg) as an output format and click on the Save button. You can save the SVG file anywhere on your local system. Open the `index.html` file with an editor of your choice and change the path to the SVG file "lift.svg" within the "data-bms-svg" attribute.

4.1.5 Start the Visualization

Let's try out the visualization for the first time! Just drag and drop the BMotionWeb manifest files on the marked area or open it via the file dialog. The visualization should start. At the top menu you will find a menu item called ProB for opening different ProB related views. For instance, Figure 4.1 shows the running lift visualization with the ProB Events view opened.

At the moment the appearance of the visualization doesn't change whenever a state change occurred (i.e. when executing events in the ProB Events view). This is because no observers exist yet. In the next Section we learn how we can link graphical elements with the formal model by establishing observers.

4.1.6 Create Observers

Observers are used to link graphical elements with the model. An observer is notified whenever the 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 graphical elements in respect to the new state. In general, observers are written in JavaScript and should be placed in the `script.js` file. As an example, consider the following *formula observer*:

```
1 bms.observe("formula", {
2   selector: "#txt_cur_floor",
3   formulas: ["cur_floor"],
4   trigger: function (origin, values) {
5     origin.text(values[0])
6   }
```

[2]The URL of the resource is: http://nightly.cobra.cs.uni-duesseldorf.de/bmotion/bmotion-prob-handbook/nightly/files/lift.svg

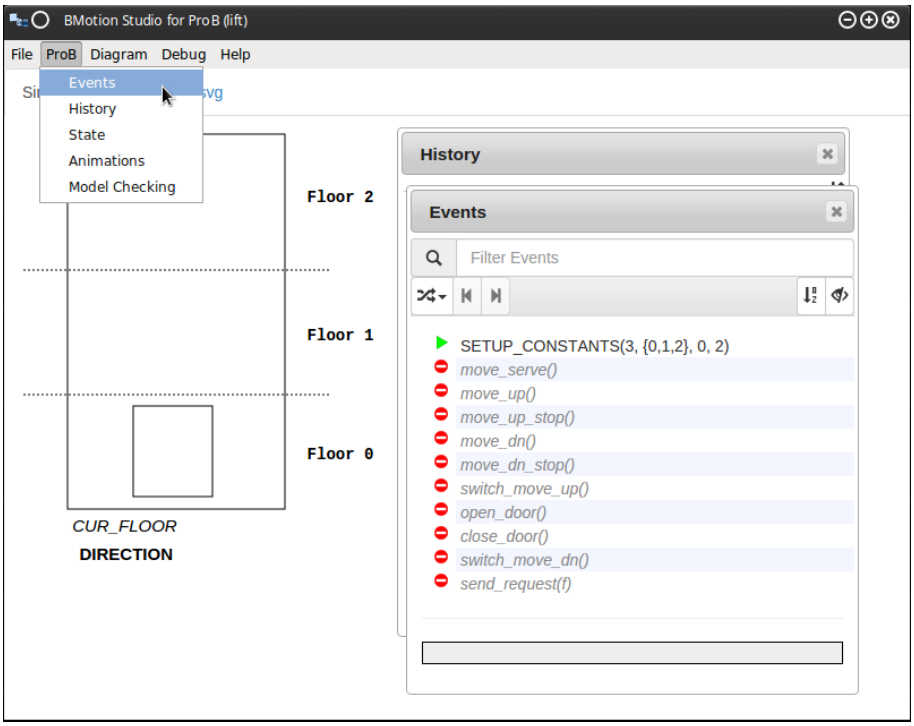

Figure 4.1: Running the Lift visualization for the First Time

```
7 });
```

Listing 4.4: Formula Observer Displaying the Current Floor (JavaScript)

Checkout Section **??** for more details about observers.

We are going to explain the JavaScript code line by line. In line 1 we register a formula observer on the graphical element with the id *txt_cur_floor* (line 2) that is located in our `index.html` file. BMotionWeb follows the jQuery selector syntax[3] to select graphical elements. The prefix "#" denotes that we want to select an element by its id. In line 3 we define a list of observed formulas. In this case we observe the variable *cur_floor*. In line 4 to 6 we define a trigger function that is called after every state change with its *origin* (the origin parameter holds a reference to the graphical element that the observer is attached to) and the *values* (the values parameter contains the values of the defined formulas in an array, e.g. use *values[0]* to obtain the value of the first formula). The trigger function changes the text of the graphical element (*origin*) to the current value of the variable *cur_floor* (*values[0]*).

Let's create another observer. Check out the following JavaScript snippet:

[3]For more information about jQuery and selectors we refer the reader to the jQuery API documentation http://api.jquery.com/category/selectors/.

```
1 bms.observe("formula", {
2   selector: "#door",
3   formulas: ["cur_floor", "door_open"],
4   trigger: function (origin, values) {
5
6     switch (values[0]) {
7       case "0":
8         origin.attr("y", "175");
9         break;
10      case "1":
11        origin.attr("y", "60");
12        break;
13      case "-1":
14        origin.attr("y", "275");
15        break;
16    }
17
18    if(values[1] === "TRUE") {
19      origin.attr("fill", "white");
20    } else {
21      origin.attr("fill", "lightgray");
22    }
23
24  }
25 });
```

Listing 4.5: Formula Observer for the Lift Door (JavaScript)

In line 1 we register a formula observer on the graphical element that matches the selector "#door" (line 2) (similar to the previous defined formula observer). In line 3 we define the set of observed formulas (*cur_floor* and *door_open*). In line 4 to 24 we define a trigger function, that makes the following action: Line 5 to 15 will switch the y coordinate of the door (denoting the movement of the door between floors) according to the current value of the variable *cur_floor* (*values[0]*). Lines 18 to 22 affect that the attribute *fill* of the door will be set to "white" (denoting the door is open) whenever the formula *door_open* evaluates to *TRUE* in the current state (*values[1]*), otherwise to "lightgray" (denoting the door is closed).

Add both snippets to your `script.js` file, save the file and click on the `Reload` button. Let's see how this affects our visualization: Setup and initialize the machine using the ProB events view. Execute some events and see what happens. For instance, Figure 4.2 shows the lift visualization where the lift is on floor 0 and the door is open.

4.1.7 Add Event Handler

In this Section we learn how we can enhance our visualization with interactive features, e.g. executing an Event-B event by clicking on a graphical element.

 Checkout Section **??** for more details about event handlers.

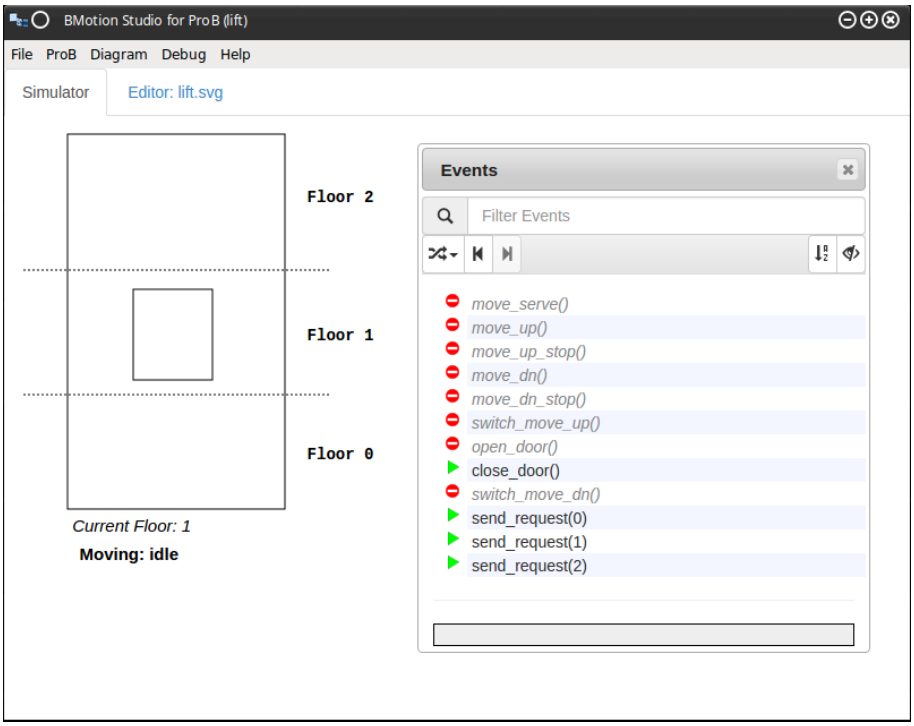

Figure 4.2: Lift visualization with observers

Let's add an interactive feature, where the user can click on a floor label to order the lift on the corresponding floor. Add this code snippet to your `script.js` file:

```
1 bms.executeEvent({
2   selector: "text[data-floor]",
3   events: [
4     {
5       name: "push_call_button",
6       predicate: function (origin) {
7         return "b=" + origin.attr("data-floor")
8       }
9     }
10  ]
11 });
```

Listing 4.6: Example of an Execute Event Handler (JavaScript)

In line 1 we register an execute event handler for each graphical element that matches the defined selector "text[data-floor]" (line 2). In particular, the selector matches the three floor labels (see Figure 4.1). In line 3 to 10 we define the list of events that should be wired with the graphical elements. Every event should contain the *name*. In addition, the user may enter a *predicate* that defines the event's arguments. If the user defines more than one event, a tooltip will be shown with a list of the defined events after clicking on the graphical element. In our example we define only one event with the *name push_call_button* and the *predicate* that is determined by a closure that passes a reference to the element (*origin*). In particular, we use the value of the attribute *data-floor* of the corresponding floor label (*origin*) to define the event parameter (line 6).

Apply these changes by clicking on the `Reload` button and try to click on a floor label. This should call the Event-B event *push_call_button* with the corresponding predicate/parameter.

Let's add another interactive feature, where the user can click on the graphical element that represents the door to open or close the door respectively. Add the following code snippet to your `script.js` file:

```
1 bms.executeEvent({
2   selector: "#door",
3   events: [
4     { name: "close_door" },
5     { name: "open_door" }
6   ]
7 });
```

Listing 4.7: Interaction with the Lift Door (JavaScript)

This execute event handler will bind to events to the graphical element that matches the selector "#door".

4.2 Observers and Interactive Handlers

BMotionWeb implements various observers and interactive handlers with different functions. They are implemented in JavaScript and follow the uniform schema shown in Listing 4.8, where *bms* is a global variable pointing to the BMotionWeb JavaScript API, *observe* a function to register an observer (see line 1) and *handler* a function to register an interactive handler (see line 2). The functions have two arguments: the first argument defines the *type* of the observer or interactive handler, and the second argument defines a list of *options* that are passed to the respective function. The options are defined as a *key/value* map, where *key* is the option's name, and *value* is the option's value. The options may be of different types (e.g. string, integer, boolean, or a function). In order to link graphical elements to observers and interactive handlers, each observer and interactive handler can define the *selector* option. The selector option determines the graphical elements to which the observer or interactive handler will be attached using the jQuery selector syntax (see Section 3.2).

```
1 bms.observe(<type>, <options>);
2 bms.handler(<type>, <options>);
```

Listing 4.8: Implementation schema for observers and interactive handlers (JS)

In the following subsections we present the various observer and interactive handler types. In each section, we first give a brief description of the characteristics of the respective observer or interactive handler and list their available options in a table. The table defines the option's name, its type, a short description and denotes if the option is required or optional. To illustrate the behavior of an observer or interactive handler, we apply it to the simple lift system presented in Fig. 3.1.

4.2.1 Formula Observer

The formula observer watches a list of formulas (e.g. expressions, predicates or single variables) and triggers a function whenever a state change occurred in the animated formal specification. The values of the formulas and the origin (the reference to the graphical element that the observer is attached to) are passed to the trigger function. Within the trigger function, the user can manipulate the origin (e.g. change its attributes) based on the values of the formulas in the respective state. Table 4.1 gives an overview of the available options for the formula observer.

Table 4.1: Available options for formula observer

Name	**Type**	**Required**	**Description**
selector	string	no	The *selector* matches a set of graphical elements which should be linked to the observer.
formulas	list	yes	A list of *formulas* (e.g. expressions, predicates or single variables) which should be evaluated in each state. For instance, $['x',' card(x)']$ observes the variable x and the expression $card(x)$ (the cardinality of the variable x).
translate	boolean	no	In general the result of the formulas will be strings. This option should be set to *true* to *translate* B-structures to JavaScript objects.
trigger	function	yes	The *trigger* function will be called after every state change with its *origin* reference set to the graphical element that the observer is linked to and with the *values* of the formulas at the new state. The *values* parameter is an array containing the values of the formulas, e.g. use *values[0]* to obtain the result of the first formula. If no selector is defined, the *trigger* function is called only with the *values* parameter.

```
1 bms.observe("formula", {
2   selector: "#door",
3   formulas: ["floor"],
4   translate: true,
5   trigger: function (origin, values) {
6     switch (values[0]) {
7       case 1: origin.attr("y", "20"); break
8       case 0: origin.attr("y", "140"); break
9       case -1: origin.attr("y", "250"); break
10     }
11   }
12 });
```

Listing 4.9: Example formula observer (JS)

Listing 4.9 shows how the formula observer is used in the simple lift system. In line 1 we register a new formula observer to the graphical element that matches the selector "#door", i.e. the graphical element that represents the door of the simple lift system (line 2). Line 3 states that the observer should observe the variable $floor$ during the animation. In line 4 we set the translate option to true. By default the results of evaluating the formulas are strings. Setting the translation option to true translates the string results into JavaScript objects. Table 4.2 gives an overview of the mapping between B (classical-B and Event-B) constructs represented as strings and JavaScript objects. For instance, the value "TRUE" is translated into the JavaScript object *true* which can be then used in the JavaScript context (e.g. in a conditional statement). In lines 5 to 11 we define a trigger function that is called whenever a state change has occurred. The reference to the matched graphical element (*origin*) and

Table 4.2: Overview of translating B constructs to JavaScript objects

		Example	
B Construct	**JavaScript**	**B as String**	**JavaScript**
BOOL	Boolean	“TRUE”	true
Naturals	Number	“2”	2
Integers	Number	“-2”	-2
Sets	Array	“{2, 3}”	[2, 3]
Sets of Sets	Array	“{{2}, {2, 3}, {2, 3, 4}}”	[[2], [2, 3], [2, 3, 4]]
Relations	Array	“{(2, 3), (3, 4)}”	[[2, 3], [3, 4]]
Nested Relations	Array	“{({(2, 3)}, 3)}”	[[[[0, 0]], 0]]
Functions	Array	“{(2, 3), (3, 4)}”	[[2, 3], [3, 4]]

the state values of the observed formulas (*values*) are passed as arguments to the trigger function. The trigger function in Listing 4.9 defines the position of the lift cabin (see lines 6 to 10). For this, it maps the y coordinate attribute of the origin to the desired value based on the state value of the $floor$ variable (*values[0]*).

4.2.2 Predicate Observer

The predicate observer observes a predicate and triggers a function depending on the evaluation of the predicate in the respective state (true or false). The reference to the graphical element to which the observer is attached is passed to the particular function. Table 4.3 gives an overview of the available options for the predicate observer.

As an example, Listing 4.10 shows a predicate observer for the simple lift system. The purpose of the observer is to set the *fill* attribute of the door to the color *white* (denoting that the door is opened) or to *gray* (denoting that the door is closed) based on to the evaluation of the predicate in the respective state (true or false). To do this, we register a new predicate observer (line 1) for the graphical element that matches the selector “#door” (line 2). In line 3 we define the predicate $door = open$ that should be observed during the animation. Lines 4 to 6 define the function that is called whenever the predicate is true in the respective state. When this is not the case the false function is called (see lines 7 to 9).

Table 4.3: Available options for predicate observer

Name	Type	Required	Description
selector	string	no	The *selector* matches a set of graphical elements which should be linked to the observer.
predicate	string	yes	A *predicate* which should be evaluated in each state.
true	function	yes	The *true* function will be called whenever the predicate evaluates to true in the respective state with its *origin* reference set to the graphical element that the observer is linked to. If no selector is defined, the *true* function is called without parameters.
false	function	yes	The *false* function will be called whenever the predicate evaluates to true in the respective state with its *origin* reference set to the graphical element that the observer is linked to. If no selector is defined, the *false* function is called without parameters.

```
1 bms.observe("predicate", {
2   selector: "#door",
3   predicate: "door = open",
4   true: function(origin) {
5     origin.attr("fill", "white");
6   },
7   false: function(origin) {
8     origin.attr("fill", "gray");
9   }
10 });
```

Listing 4.10: Example predicate observer (JS)

4.2.3 Set Observer

The state-based formal methods classical-B and Event-B are based on set theory. Thus, the different aspects of the system are often expressed as sets. As an example, consider a formal specification of an interlocking system, where the occupied block segments of a track are expressed as a set. It would be useful to identify graphical elements based on the elements of this set and to color all of them red at once (denoting that the blocks are occupied). To do this, we present an observer called *set observer* that is capable of selecting graphical elements based on a user-defined set expression. Table 4.4 shows the available options for the set observer.

To illustrate the use of the set observer consider Listing 4.11. The purpose of the observer is to set the *fill* of all pressed request buttons to *green*. To do this, we define the set selector based on the variable *request* which defines the set of floor numbers where the request button has been pressed (line 3). Since the ids of the graphical elements that represent the request

Table 4.4: Available options for set observer

Name	Type	Required	Description
selector	string	no	The *selector* matches a set of graphical elements which should be linked to the observer.
set	string	yes	The result of the defined *set* expression is used to establish a *set selector* which in turn is used to find child graphical elements of the graphical element that matches the *selector* of the observer. The elements of the set are joined with the prefix "#" (e.g. "#ele1,#ele2,#ele3,...").
convert	function	no	The *convert* function is called for each element of the defined set. It returns an element selector of the form "#id", where *id* is the identifier of the element. The user can also override the method.
actions	list	yes	A list of *actions* that determine the appearance and the behaviour of the *set* graphical elements.
attr	string	yes	The *attribute* of the elements that should be modified.
value	string	yes	The new *value* of the attribute.

buttons have the form "bt_*nr*", where *nr* is the respective floor number (−1, 0 or 1), we override the prefix using the convert function (lines 4 to 6). The returned prefix is composed of the string "#bt_" and the floor number (e.g. "#bt_0"). Finally, in lines 7 to 10 we define the *actions* triggered on the graphical elements that matches the composed *set* selector: we color the graphical elements in *green* (denoting the buttons that are pressed).

```
1 bms.observe("set", {
2   selector: "#request_buttons",
3   set: "request",
4   convert: function(element) {
5    return "#bt_" + element;
6   },
7   actions: [{
8    attr: "fill",
9    value: "green"
10  }]
11 });
```

Listing 4.11: Example set observer (JS)

4.2.4 Refinement Observer

Refinement is an important concept in the state-based formal methods classical-B and Event-B. It can be used to structure the development of a formal specification and to gradually introduce complexity and details (e.g. new variables or events). In order to support refinement in interactive formal prototypes, we introduce an appropriate observer with the

Table 4.5: Available options for refinement observer

Name	Type	Required	Description
selector	string	no	The *selector* matches a set of graphical elements which should be linked to the observer.
refinement	string	yes	The *refinement* that should be observed. The option accepts the name of a classical-B or Event-B machine.
enable	function	yes	The *enable* function is called whenever the defined *refinement* is part of the animation with its origin reference set to the graphical element that the observer is linked to. If no selector is defined, the *enable* function is called without parameters.
disable	function	yes	The *disable* function is called whenever the defined *refinement* is not part of the animation with its origin reference set to the graphical element that the observer is linked to. If no selector is defined, the *disable* function is called without parameters.

available options shown in Table 4.5.

Listing 4.12 shows the use of the refinement observer based on the Event-B simple lift system. The purpose of the observer is to show the request buttons of the visualization (see Fig. 3.1) only if the corresponding refinement (the machine $m2$ where the buttons are introduced) is part of the animation, otherwise the request buttons should be hidden. To do this, we register a new refinement observer to the group of request button graphical elements ("#request_buttons"). In line 3 we define the refinement (the name of the machine) that introduces the request buttons: $m2$. Lines 4 to 6 define the *enable* function that sets the *opacity* attribute of the graphical element to the value *1* (showing the request buttons) whenever the defined refinement is part of the animation. Otherwise, the *disable* function (lines 7 to 9) is called which sets the *opacity* attribute of the graphical element to the value *0* (hiding the request buttons).

```
1 bms.observe("refinement", {
2   selector: "#request_buttons",
3   refinement: "m2",
4   enable: function (origin) {
5     origin.attr("opacity", "1")
6   },
7   disable: function (origin) {
8     origin.attr("opacity", "0")
9   }
10 });
```

Listing 4.12: Example refinement observer (JS)

4.2.5 Illustration of Observers

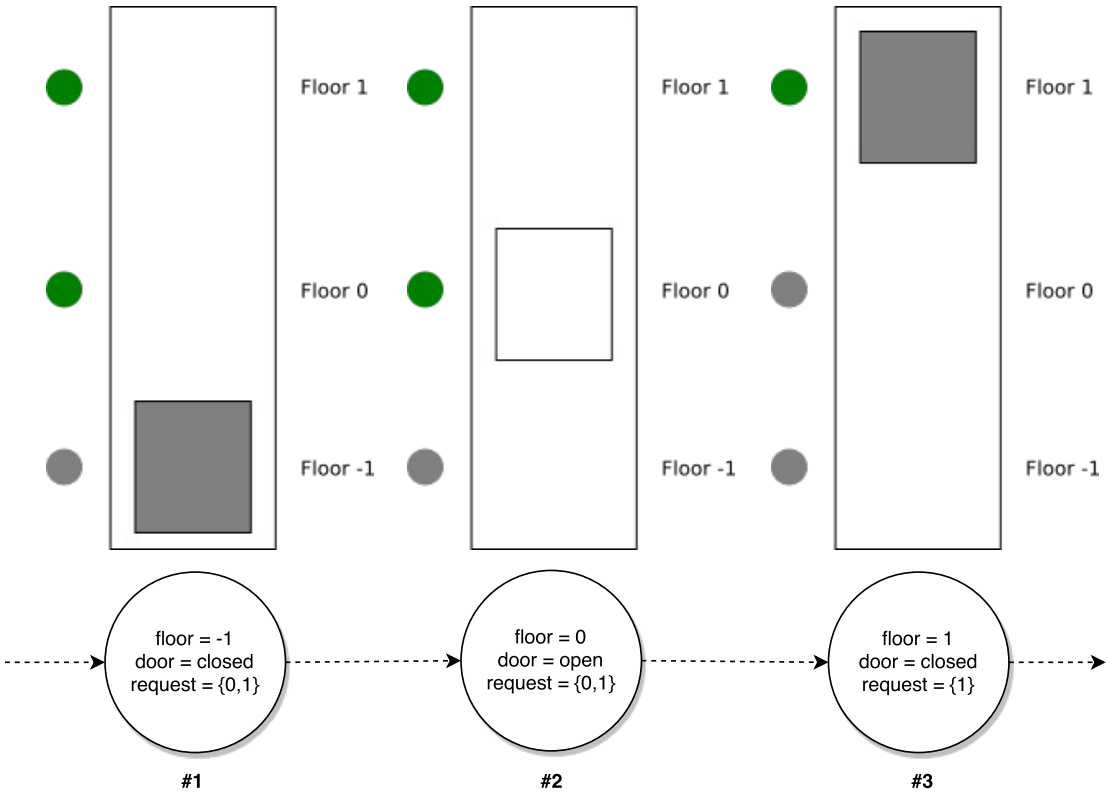

Figure 4.3: Effect of formula, predicate and set observers on simple lift system

Figure 4.3 illustrates the effect of the example formula (Listing 4.9), predicate (Listing 4.10) and set (Listing 4.11) observers on the simple lift system (Fig. 3.1). Some example states and their variable configurations are shown at the bottom of the figure. The effect of applying the observers is shown at the top of the figure. As can be seen in the figure, the effect of the formula observer is to change the y coordinate based on the current state value of the variable $floor$ (denoting the movement of the door between floors). The effect of the predicate observer is to set the *fill* color of the lift door according to the evaluation of the predicate $door = open$. For instance, in state *#2* the predicate is $true$. Hence, the door is *white* denoting the door is opened. Finally, the set observer colors all pressed request buttons to *green* based on the set variable *request*.

4.2.6 Execute Event Handler

The execute event handler wires a list of classical-B operations or Event-B events to graphical elements. Table 4.6 shows the available options for the execute event handler.

Listing 4.13 shows how the execute event handler is used. In line 1, we register a new execute event handler for the graphical element that represents the request button for floor

Table 4.6: Available options for execute event handler

Name	Type	Required	Description
selector	string	yes	The *selector* matches a set of graphical elements which should be linked to the interactive handler.
events	list	yes	A list of *events* which should be wired with the graphical element.
name	string	yes	The *name* of the event.
predicate	string	no	The *predicate* for the event.
label	function	no	The *label* function returns a custom label as a string to be shown in the tooltip. The user can also return an HTML element. The function provides two arguments: the *origin* reference set to the graphical element to which the handler is linked and the *event* data.
callback	function	no	The *callback* function will be called after the event has been executed. If the event returns a value (e.g. when executing a classical-B operation with return value), the return *value* is passed to the *callback* function.

0 (line 2). In lines 3 to 8, we define the event with the event's *name* (line 5) and *predicate*[4] (line 6) which should be wired to the graphical element. Finally, in lines 9 to 11 we define a custom label based on the data of the *event* object which contains the name (event.name) and the predicate (event.predicate) of the defined event.

```
1 bms.handler("executeEvent", {
2   selector: "#bt_0",
3   events: [
4     {
5       name: "send_request",
6       predicate: "f=0"
7     }
8   ],
9   label: function(origin, event) {
10     return "Push button " + event.predicate;
11   }
12 });
```

Listing 4.13: Example execute event handler (JS)

Figure 4.4 illustrates the effect of the execute event handler. A tooltip that lists all available events (disabled and enabled) will be shown when hovering over the graphical element or when clicking on the graphical element and if all events are disabled or more than one event is enabled. If only one event is enabled, it is executed directly when clicking on the graphical element. As an example, in the figure the user hovers over the request button

[4]The predicate defines the values of the parameters for the event.

on floor 0.

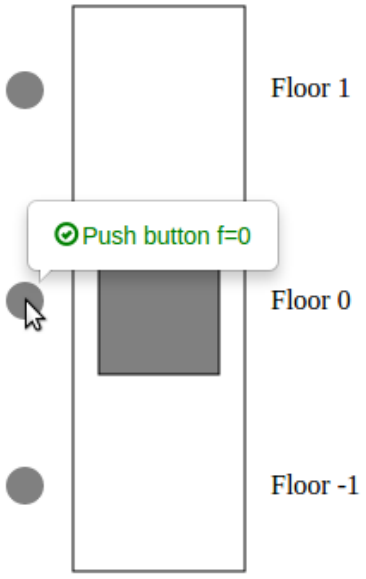

Figure 4.4: Effect of execute event handler on simple lift system

4.2.7 Context-Sensitive Options

Each option for an observer or interactive handler (except of the selector option and the options that take a function as its value) can also define a function that returns its value. The *origin* (the reference to the graphical element that the observer or interactive handler is attached to) is passed to the value function as the first parameter. Defining a value function enables the user to determine the value of an option in the context of the linked graphical element. As an example, consider the execute event handler presented in Listing 4.13. The handler wires the *send_request* event with the predicate $f = 0$ to the graphical element that represents the lift request button on the floor 0 (*#bt_0*). Instead of creating similar execute event handlers for the other request buttons, we could also define a selector that selects all request buttons and a value function that returns the predicate in context of the matched graphical elements. Listing 4.14 shows an alternative execute event handler linked to all ellipse graphical elements that provide a *data-floor* attribute (*ellipse[data-floor]*). The *data-floor* attribute defines the floor number (-1, 0 or 1) of the respective request button. In line 6 to 8 we define a function that returns the predicate of the event *send_request* in context of the matched graphical elements, i.e. the function returns the predicate based on the *data-floor* attribute of the linked graphical element. For instance, the predicate function returns $f = 1$ for the graphical element where the *data-floor* attribute is set to 1. Based on context-sensitive options, we can create generic observers and interactive handlers: if we add more request floor buttons, the execute event handler in Listing 4.14 would be also valid for the new buttons.

```
bms.executeEvent({
  selector: "ellipse[data-floor]",
  events: [
    {
      name: "send_request",
      predicate: function (origin) {
        return "f=" + origin.attr("data-floor")
      }
    }
  ],
  label: function(origin, event) {
     return "Push button " + event.predicate;
  }
});
```

Listing 4.14: Context sensitive execute event handler (JS)

4.2.8 Other API Features

The BMotionWeb JavaScript API also provides some other features listed below:

Evaluate formulas manually. The JavaScript API provides the *bms.eval* function that takes a list of options defining the *formulas* to be evaluated and a *trigger* function that is called with the values of the formulas. The function is similar to the *formula observer*, except that the *bms.eval* function is executed once (in the current state) rather than after every state change.

Execute transitions manually. With the *bms.executeEvent* API function, the developer can execute a transition manually. The function takes a list of options, where the name and predicate options define the name and the predicate of the event to be executed respectively. Similar to the *execute event handler* the developer can optionally define a callback function that is called after the event has been executed. If the event returns a value (e.g. for a classical-B operation with a return value) the return *value* is passed to the *callback* function.

Initialization listener. The *bms.init* function takes a function as its parameter that is called whenever the animated formal specification is initialized. Thus, the developer could create the visualization according to static data coming from the formal specification (e.g. constants or external data from a database).

4.3 External Method Calls

BMotionWeb provides a Groovy API that can be accessed via the global variable *bms* within a Groovy script file. The Groovy API provides different functions to programatically control the integrated animation engine and to interact with the animated formal specification, e.g.

to access the state space or trace of the animated formal specification. The developer can also register external methods that are evaluated on the server side. The registered methods accept arguments from the client and may also return data to the client. Listing 4.15 demonstrates the *bms.registerMethod* Groovy API function.[5] The method takes two arguments: the first argument defines the name under which the method should be registered, and the second argument is a closure that defines the actual method. For instance, in line 1 we register a method called *random* with a parameter n and the method body defined in lines 2 to 11. The purpose for this method is to randomly execute n events in the animated formal specification, where n is a number passed to the method. If a number below or equal zero has been passed to the method the method returns an error message. Otherwise the method randomly executes the event and returns a success message.

Since BMotionWeb integrates with the ProB animation engine, some of the ProB functionality is exposed to the user via the BMotionWeb Groovy API. ProB is tightly integrated with the Groovy scripting language. Everything from the constraint solver to the user interface is exposed via the scripting language.[6] For instance, in lines 5 and 7 in Listing 4.15 we access the current trace (*bms.getTrace()*) and execute a random event (*trace.anyEvent()*) of the animated formal specification respectively. These methods then call the appropriate methods within the ProB Java API.

```
1 bms.registerMethod("random", { n ->
2  if(n <= 0) {
3   return "Only numbers greater than 0 are allowed.";
4  } else {
5   def trace = bms.getTrace();
6   1.upto(n, {
7    trace = trace.anyEvent();
8   });
9   bms.getAnimationSelector().traceChange(trace);
10  return n + " events have been executed.";
11 }
12 });
```

Listing 4.15: Register method on the server side (Groovy)

To use a registered method on the client side, the method can be wired to graphical elements (e.g. with an observer or interactive handler) or the developer can call the method manually (see lines 11 to 18 in Listing 4.16). Lines 2 to 9 in Listing 4.16 demonstrate the *execute method handler* that executes a registered server side method when the user clicks on the linked graphical element. In line 2 we register the handler on the graphical element that matches the selector *#button* (line 3). In line 4 and 5 we define the *name* and *args* (the arguments that should be passed to the method) of the method to be called. In lines 6 to 8 we define a *callback* function that is called whenever the method on the server side returns a value. The *origin* (the reference to the graphical element) and the returned *data*

[5] An overview of the BMotionWeb Groovy API functions is given in Section 4.3.1.

[6] A documentation of the ProB Java API is available at http://www3.hhu.de/stups/handbook/prob2/current/devel/html.

Table 4.7: Available options for method observer and execute method handler

Name	Type	Required	Description
selector	string	no (observer) yes (handler)	The *selector* matches a set of graphical elements which should be linked to the observer or handler.
name	string	yes	The *name* of the registered server side method.
args	list	no	The *args* that should be passed to the registered server side method.
callback	function	no	The *callback* function is called whenever the server side method returns a value with its origin reference set to the graphical element that the observer or handler is linked to and the return *value* of the method.

is passed to the callback function. In a similar fashion, an observer can be defined for observing a registered server side method (i.e. the method is called after every state change). Table 4.7 gives an overview of the available options for the method observer and execute method handler.

```
1 // Register execute method handler
2 bms.handler("method", {
3   selector: "#button",
4   name: "random",
5   args: [10],
6   callback: function(origin, data) {
7    alert(data);
8   }
9 });
10
11 // Call method on server side manually
12 bms.callMethod({
13  name: "random",
14  args: [10],
15  callback: function(msg) {
16   alert(msg);
17  }
18 });
```

Listing 4.16: Use registered method on client side (JS)

4.3.1 BMotionWeb Groovy Scripting API

```
1  package de.bmotion.core;
2
3 import java.util.Map;
4
5 import groovy.lang.Closure;
6
```

```
7  public interface IBMotionApi {
8   /**
9    *
10   * Logs the given message on the client side. An arbitrary object can be
11   * passed as a message with the assumption that the object is serializable.
12   *
13   * @param message
14   * An arbitrary serializable message object
15   */
16  public void log(Object message);
17
18  /**
19   *
20   * Executes an event for the given name.
21   *
22   * @param name
23   * The name of the event that should be executed
24   * @return The return value of the event (e.g. classical-B operations may
25   * have return values)
26   * @throws BMotionException
27   */
28  public Object executeEvent(String name) throws BMotionException;
29
30  /**
31   *
32   * Executes an event for the given name and options.
33   *
34   * @param name
35   * The name of the event that should be executed
36   * @param options
37   * The options for the event (e.g. an additional predicate)
38   * @return The return value of the event (e.g. classical-B operations may
39   * have return values)
40   * @throws BMotionException
41   */
42  public Object executeEvent(String name, Map<String, String> options) throws
        BMotionException;
43
44  /**
45   *
46   * Evaluates the given formula in the current state and returns the value.
47   *
48   * @param formula
49   * The formula that should be evaluated in the current state
50   * @return The result of the formula
51   * @throws BMotionException
52   */
53  public Object eval(String formula) throws BMotionException;
54
55  /**
56   *
57   * Evaluates the given formula with options in the current state and returns
```

```
58   * the value.
59   *
60   * @param formula
61   * The formula that should be evaluated in the current state
62   * @param options
63   * The options for the evaluation (e.g. translate flag)
64   * @return The result of the formula
65   * @throws BMotionException
66   */
67  public Object eval(String formula, Map<String, Object> options) throws
         BMotionException;
68
69  /**
70   *
71   * Registers a method on the server side.
72   *
73   * @param name
74   * The name of the method.
75   * @param func
76   * The functional body of the method as a {@link Closure}
77   */
78  public void registerMethod(String name, Closure<?> func);
79
80  /**
81   *
82   * Calls a registered method on the server side.
83   *
84   * @param name
85   * The name of the method.
86   * @param args
87   * The arguments for the method
88   * @return The return value of the method
89   * @throws BMotionException
90   */
91  public Object callMethod(String name, Object... args) throws BMotionException;
92
93  /**
94   *
95   * Returns a list of registered server side methods.
96   *
97   * @return A list of registered server side methods
98   */
99  public Map<String, Closure<?>> getMethods();
100
101 /**
102  *
103  * Returns session related data.
104  *
105  * @return Session related data
106  */
107 public Map<String, Object> getSessionData();
108
```

```
109  /**
110   *
111   * Returns tool related data.
112   *
113   * @return Tool related data
114   */
115  public Map<String, Object> getToolData();
116
117 }
```

Listing 4.17: BMotionWeb Groovy Scripting API

```
1  package de.bmotion.prob;
2
3  import de.bmotion.core.IBMotionApi;
4  import de.prob.model.representation.AbstractModel;
5  import de.prob.statespace.AnimationSelector;
6  import de.prob.statespace.Trace;
7
8  public interface IProBVisualizationApi extends IBMotionApi {
9
10  /**
11   *
12   * Returns the ProB representation of the loaded formal specification.
13   *
14   * @return The formal specification as {@link AbstractModel}
15   */
16  public AbstractModel getModel();
17
18  /**
19   *
20   * Returns the current {@link Trace} of the animation.
21   *
22   * @return The current {@link Trace} of the animation
23   */
24  public Trace getTrace();
25
26  /**
27   *
28   * Returns the {@link AnimationSelector} which is the entry point to the
29   * ProB GUI.
30   *
31   * @return The {@link AnimationSelector} which is the entry point to the
32   * ProB GUI
33   */
34  public AnimationSelector getAnimationSelector();
35
36 }
```

Listing 4.18: BMotionWeb for ProB specific Groovy Scripting API

4.4 Visual Editor

An important component of BMotionWeb is the built-in visual editor. The overall goal of the editor is to facilitate the rapid creation of visualization templates. The editor has been implemented and adapted based on *method draw*, a web based SVG editor.[7] Figure 4.5 shows the editor while editing the visualization template of the simple lift system interactive formal prototype. The editor consists of a palette for creating graphical elements, like shapes, labels, and images and a view for managing the properties of graphical elements. Graphical elements can be added to a canvas which provides like drag and drop, undo/redo, copy/paste and zooming.

Figure 4.5: Built-in visual editor of BMotionWeb

[7] https://github.com/duopixel/Method-Draw.

The visual editor also supports the creation of observers and interactive handlers. Two additional views (one for creating observers and a second for creating interactive handlers) are available for this purpose. As an example, Fig. 4.6 shows the observers view. The view lists all observers with their corresponding options for the current edited visualization template. The user can edit the options of an observer directly in the observers view. If an option has a JavaScript function as its value, a JavaScript editor is shown when editing the option. For instance, the left side of Fig. 4.6 shows the JavaScript editor for the trigger function of the formula observer that is wired to the graphical element *#door*. The user only needs to provide the body of the function. The arguments (*origin* and *values*) are passed directly to the function body while running the interactive formal prototype.

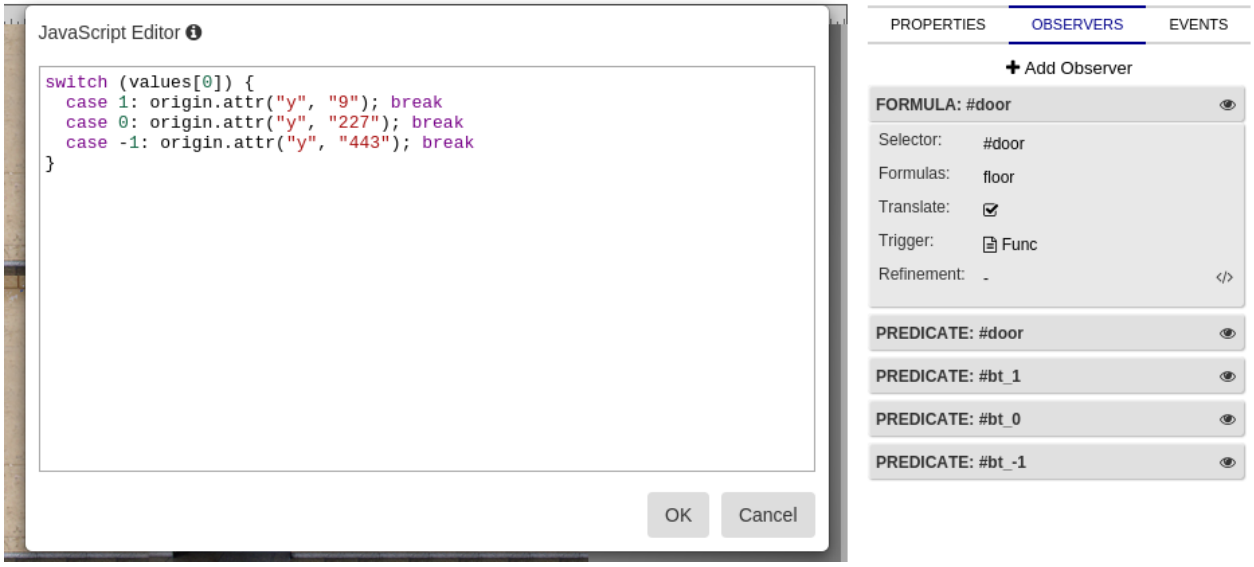

Figure 4.6: Observers view in visual editor of BMotionWeb

Chapter 5

Frequently Asked Questions

5.1 Where can I download the tool?

You can find the latest version of the tool at http://www.stups.hhu.de/ProB/index.php5/BMotionWeb_Download.

5.2 Where can I report bugs?

If you want to submit a bug report, please use our bug tracker. You may also want to ask questions within our prob-users group.

5.3 Where can I find examples?

You can find a bunch of examples at GitHub: https://github.com/ladenberger/bmotion-prob-examples.