INSTITUT FÜR INFORMATIK
Lehrstuhl für Softwaretechnik und
Programmiersprachen

Universitätsstr. 1 D–40225
Düsseldorf

A Visualisation Plugin for ProB2-UI

Michelle Angela Werth

Bachelorarbeit

Beginn der Arbeit:	21. März 2019
Abgabe der Arbeit:	21. Juni 2019
Gutachter:	Prof. Dr. Leuschel
	Dr. Bendisposto

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 21. Juni 2019

Michelle Angela Werth

Abstract

The modern economy depends on safety-critical systems daily. These systems ought to work autonomously and have to undergo a lot of verification in development. For this, formal methods experts create models based on real-world machines and systems.

Most of the time, a requirement for the development of formal models is that domain experts and formal methods experts work together. Visualisations simplify the work with formal models, e.g. developed with the B-Method. B models can be developed without a visualisation, but using visualisation can improve the communication between formal methods experts and domain experts. Additionally, the B model can be visualised similarly to the real-world situation. That makes it easier to imagine the specification of the B model.

ProB is an animator and model checker for formal models and has been developed to simplify the work with B models. But the usability of ProB itself can be improved. That is why ProB2-UI was built on top of it. ProB2-UI gives B developers the possibility to animate, model check, and constraint solve their B models in a user-friendly interface.

This thesis presents VisB, a visualisation plugin for ProB2-UI. This plugin enables the user to create simple visualisations for B models. VisB is based on Java, JavaFX, and JavaScript. The tool uses an SVG image and a visualisation specification file, in short, VisB file to generate a visualisation for the B model.

Moreover, the thesis investigates the goals of usability, maintenance, and capability of creating a broad range of visualisations for VisB. Additionally, it is shown how VisB was developed and why it is useful for ProB2-UI.

It is then presented how the user interface is made user-friendly while considering the extensiveness of the plugin.

Later on, the thesis examines how to create a visualisation with VisB in two simple visualisation examples. The visualised B models are a lift and the n-queens problem. This thesis also contains use cases for each of these examples.

There have been several visualisation tools for ProB in the past. The *ANIMATION_FUNCTION* [21], *BMotionWeb* [18] and the *visualisation mechanism* [13] are investigated in regards to the focus of this thesis. It is explored why these tools either cannot be maintained, are not user-friendly enough or cannot visualise a broad range of B models.

Contents

1 Introduction and Motivation

Our modern society fully depends on machines and autonomous systems. Most big companies rely on those machines and systems to produce their goods, transport their products to sellers, and even manage their employees. To make all of that possible, those systems need to be reliable. They have to undergo a lot of testing and verification before one can think of fabricating or implementing them. But how does one verify that something is working correctly if it cannot be manually tested, yet?

One can do that by constructing a formal model of these real-life problems, which can be verified virtually. This can be done with the B-Method [3], in which those models are called B machines. Visualisation and animation tools are needed because it is hard to imagine, what a formal construct would act like in a real-world situation and some properties of such models are easier explored with visualisation and animation.

Visualisation of a model is very important for formal methods experts and domain experts. Domain experts do not need knowledge of formal methods to understand their domain. Formal methods experts can visualise models and discuss the correctness of a model with domain experts in that way. In this thesis, a visualisation plugin for ProB2-UI is implemented, with which one can create visualisations for B models.

Because the ProB2 team consists of students, doctors, and professors, its members inevitably change over time. Furthermore, students often have not a lot of experience in developing software. Therefore, one has to construct maintainable software. Hence, in this thesis, the focus lays on how the project can be maintained easily.

An intuitive user interface is very important for this project. The ProB2 team has been working with companies and organisations, like Thales [12], Alstom [8] and more. That means, for this plugin to be helpful, it should enable the user, the formal methods expert, to visualise B models easily. In section 4, it is further explained, how one can achieve an intuitive design.

A visualisation tool should cover a broad range of B models which can be visualised with it. How this plugin is covering almost everything the user needs for constructing a visualisation for most models, is explained in section 5.

ProB2-UI has already a lot of features, which allow developers to specify and verify their B model in a comprehensible way. In this thesis, VisB is introduced, a new visualisation plugin for ProB2-UI and it is shown how and why VisB is an improvement for ProB2-UI.

One goal for VisB is to build something similar to *BMotionWeb* [18], which is further explained in subsection 6.2. *BMotionWeb* is a very sophisticated visualisation tool for B machines which additionally allows validation of these visualisations [18]. It is based on web-technologies, like JavaScript, HTML and SVG [18]. Because *BMotionWeb* is not maintained any longer, one of the goals of this thesis is to build a likewise intuitive and extensive visualisation tool that can be maintained more easily.

Next, VisB should have more unrestricted visualisation possibilities than the *ANIMA-*

TION_FUNCTION [21]. The *ANIMATION_FUNCTION* is implemented in ProB and to use it one directly writes in the B model code. This tool is explained in more detail in subsection 6.1.

Finally, it should be easier to construct a working visualisation for ProB2 users, without a broad knowledge of other programming languages. This is a problem when using the *visualisation mechanism* [13]. This *visualisation mechanism* is a tool, with which a formal methods expert has to write Java and JavaFX code to construct a visualisation [13]. The subsection 6.1 explains this tool as well.

In short, the three goals of this thesis are to build a visualisation plugin which is easy to use, easy to maintain, and enables the user to visualise a wide range of B models. Additionally, it is a set requirement to use SVG images as a base for the visualisation, because this file format is easy to understand and use.

VisB uses JavaScript to interact with an image file and a visualisation specification file to create a visualisation for a B model. Different approaches to the development of the architecture will be demonstrated. Moreover, this thesis explains how VisB is made more maintainable than *BMotionWeb*. Additionally, it describes why this plugin is useful for B developers and domain experts. It is answered, how one can achieve an intuitive user interface with simple design choices. After that, it is shown how VisB can be used for visualising B models in two examples. Finally, VisB is compared to previous visualisation possibilities for ProB and ProB2-UI, in consideration of the previously mentioned goals.

2 Fundamentals

This section describes the B-Method [3], ProB2 [17], and ProB2-UI [22]. These are three significant keywords that have to be understood when reading further into this thesis.

2.1 B-Method

In 1980 Jean-Raymond Abrial developed the B-Method [3]. The B-Method is a formal method to specify and verify hardware and software systems with the B language. Models created by the B language are called B models. These B models represent a machine or a system that formal method experts specify.

"The kind of systems we are interested in developing are complex and discrete." [1] The author means that formal methods can be used to inspect a model that operates autonomously. Equally important is that a model "requires a high amount of correctness" because it operates in unpredictable environments [1].

2.2 ProB2 and ProB2-UI

"ProB is an animator, constraint solver and model checker for the B-Method." (Introduction) [22] ProB was introduced in 2003, by Leuschel and Butler [7]. With this tool, one can verify a system, based on the B-Method, in development. In ProB it is also possible to work with other verification languages, like Alloy [16], TLA+ [20], Event-B [2], Z [23], and more [22].

After the development of ProB, the ProB team implemented the ProB 2.0 Java API [17] to simplify the work with B models. The intention of implementing the ProB 2.0 Java API was to build a user interface on top of the API [22]. This user interface, ProB2-UI, was successfully implemented in JavaFX. Many additional features, plugins, and other tools are developed for ProB2-UI to further enhance the user experience.

3 Architecture

The development of the architecture is one of the most important steps for building software. The architecture, in this case, accounts for the maintainability and the wide-ranged possibilities of the visualisation plugin. Consequentially, the architecture of VisB has to undergo a lot of thought in development.

The following describes the development of the architecture of VisB. Different approaches are analysed and it is explained how the current architecture of VisB is achieved. After that, the questions are answered which input files are needed and why they are needed. The following sections also explain why this architecture is maintainable. Finally, it is explained why the current implementation of VisB is a possible solution to satisfy the goals of this thesis.

3.1 Development of the Architecture

This section investigates the development of VisB's architecture. Furthermore, it explains why the following approaches are not suitable for this project. In the end, the final approach for the current architecture is described.

3.1.1 Approach with JavaFX and SVGPath

SVGPath is a class in the JavaFX libraries that enables the developer to visualise an SVG image in JavaFX and make it interactive through JavaFX intern controls. In 2018 Heinzen introduced the *visualisation mechanism* [13]. With this *visualisation mechanism*, the user has to write Java code with the JavaFX class SVGPath to visualise a B model. A formal methods expert, therefore, has to understand Java and JavaFX to visualise with this tool. Moreover, the visualisation with SVGPath and JavaFX can be difficult and intricate. Because of that, the idea is to simplify the process of creating a visualisation when using the *visualisation mechanism*.

One of the goals for this thesis is to use SVG to create a visualisation. When using the *visualisation mechanism* [13] one, therefore, has to find a way to interact with an SVG image inside of this tool. The solution for this is to translate SVG images to the JavaFX class, SVGPath, used in *visualisation mechanism*.

One can use pre-made SVG images and visualisation specification to create a visualisation for the *visualisation mechanism* [13]. For that, one has to use those input files and translate them to a JavaFX project, as one can see in Figure 1. This project can then be executed with the *visualisation mechanism*. The JavaFX project can contain more than one JavaFX file, as one can see in visualisations done manually by Heinzen [13].

In short, for this approach, one has to implement a translator which translates SVG images to a JavaFX project and a user interface for this translator.

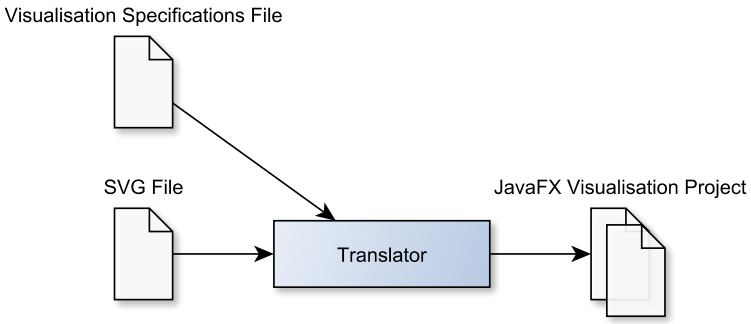


Figure 1: Using a SVG to JavaFX

The JavaFX visualisation project in Figure 1 represents the final visualisation that can be executed with Heinzen's *visualisation mechanism* [13]. That means displaying the visualisation is already implemented in ProB2-UI because the *visualisation mechanism* is implemented in ProB2-UI. Therefore, this approach has the advantage that the focus of this work could lay on the translator and the user interface for VisB.

The *visualisation mechanism* is written in Java and JavaFX [13]. Consequently, one would not have to use language support interfaces in the implementation. This simplifies the architecture.

Translating an SVG image to SVGPath, however, has its disadvantages. The SVGPath class is limited to the Java stack and one is not able to translate every SVG specification to it. Additionally, there are limited possibilities of what the SVGPath class is capable of. E.g. animations are something the SVGPath class is not capable of doing without hard-coded Java methods and objects. For these, the user would have to write Java code again because those are specific for one visualisation.

Moreover, there would be one JavaFX project for each visualisation, which has to be maintained or rewritten after updates to the translator or ProB2-UI. A system like that would be difficult to maintain.

All in all, the approach to implement a translator, which translates SVG images into executable visualisations is neither maintainable nor useful enough to further develop it. This is why further ideas are needed.

3.1.2 Approach with JavaScript

JavaScript [11] is used for interactive web applications and works well with SVG. In web browsers, SVG images can be controlled via jQuery library calls and CSS. Hence, the next idea is to solely use JavaScript to write the visualisation tool and to use its capabilities to improve the interaction with the SVG file.

The idea with this is to implement a web application that works inside of the JavaFX web view as shown in Figure 2.

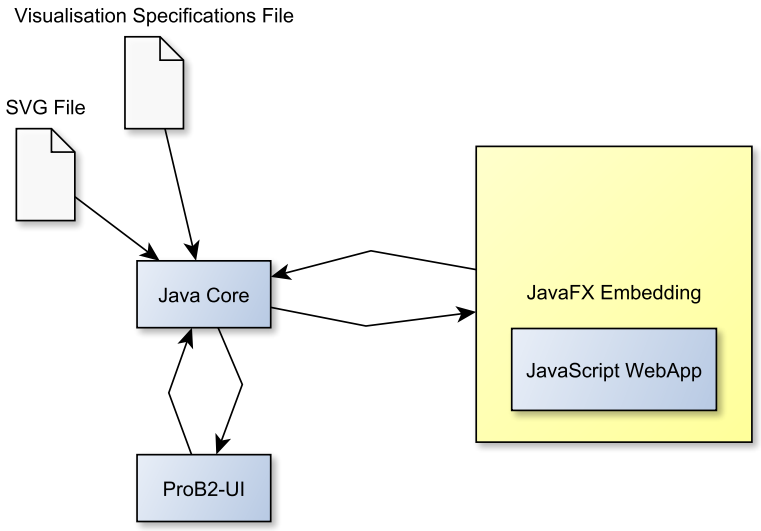


Figure 2: Using JavaScript

As seen in Figure 2, to implement this idea, one has to build a Java Core that coordinates the interaction between the JavaScript web application and ProB2-UI. One would need the SVG file and a visualisation specification file for this as well. The visualisation file for this web application would contain the information needed to modify the SVG image.

With the last approach, the user would have to write a bit of Java code for specific animations or interactions. It would not be necessary for the user to write anything else than the visualisation specification file and the SVG file. In this approach, it is easier to separate specific visualisation aspects from the Java code, which makes the visualisation specification more general. This is easier to maintain than generated JavaFX classes.

The advantages of this approach are that it would be easier for the user to create vi-

sualisations. The implementation of the interaction with the SVG image would be uncomplicated because web technologies like JavaScript and CSS have built-in libraries to work with SVG images. That means one could use the exact SVG image from the user's input to visualise the B model.

Nevertheless, the maintainability of the tool would have decreased by using this approach because one has to develop a lot of code in JavaScript. Implementing a similar approach, *BMotionWeb* [18] has been done before and it could not be maintained.

In summary, the idea of using an embedded JavaScript web application is easy to use and additionally fulfils the goal that a wide range of visualisations is possible. However, this tool would be very difficult to develop and to maintain.

3.1.3 Approach with JavaFX and JavaScript

This last idea is seemingly the best strategy to meet all the goals of this thesis. This is because it uses all the advantages of previous ideas and combines them.

At first, it seems difficult to combine the JavaScript visualisation and interaction possibilities with the rather limited ones in JavaFX. However, a very understandable chart of the architecture can be constructed, as shown in Figure 3.

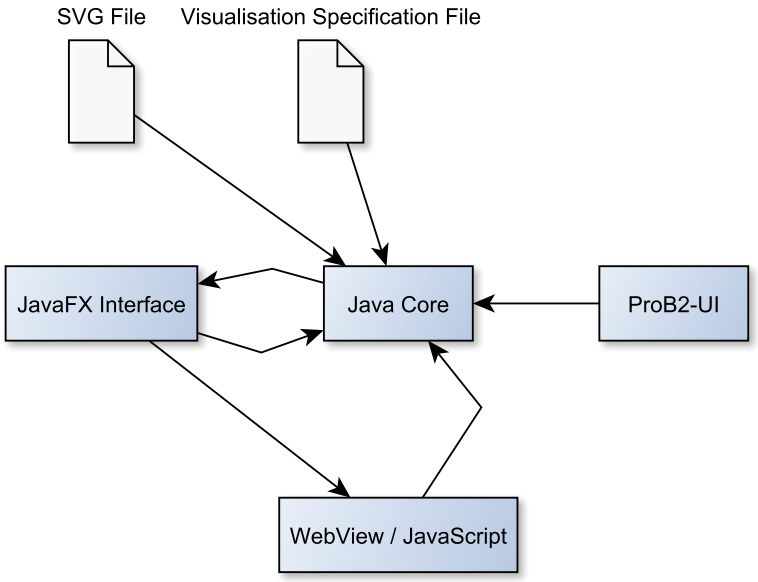


Figure 3: Using JavaFX and JavaScript

This architecture consists of every positive feature of previous ideas built into one visualisation plugin. To implement it, the concept is to build a Java Core. That Java Core interacts with a JavaFX interface for the user interface. That user interface interacts with its web view in which JavaScript can be used to manipulate the SVG

image.

The Java Core can process the SVG and the visualisation specification file to form a visualisation. This visualisation is visible in the web view. Additionally, the Java Core interacts with the ProB2-UI to execute operations and possibly get information about the B model.

The JavaFX interface further interacts with the built-in web view, which manipulates the SVG image via JavaScript and jQuery library calls. By using Heinzen's plugin mechanism [13], one can focus on implementing a maintainable Java Core and a user-friendly JavaFX-UI.

One advantage of this approach is that the implementation of it is mainly in Java and JavaFX. That means one can keep the JavaScript part as small as possible, to ensure an easy to maintain software. In addition to that, the UI could be implemented in consideration of the ProB2-UI, so that the user can easily switch between using in both. And the last point, a wide variety of visualisations without too much trouble for the user, can be implemented with this as well.

Because this architecture is based around a visualisation specification file, which one can see in Figure 3, it is possible the user has to write a lot of specifications for a visualisation to work as intended. But this be countered by further developing the plugin, so that it works as an editor for visualisation specification files as well, or by limiting the possibilities of the visualisation plugin. A possible solution to this problem is shown in section 8.

Finally, it is decided to build VisB based on this architecture, because it fulfils the goals that were established before: usability, maintainability, and a wide variety of possible visualisations.

3.2 Input Files

After figuring out the architecture, one has to think about the visualisation specification file and the SVG image file which were mentioned in the previous chapters.

With the usage of input files, one can create visualisations that are separate from the B models code and general enough to allow good maintainability. This makes the B model more readable and the overall visualisation tool more useful. If needed, one can use the B model separately from its visualisation and it is possible to create one visualisation for multiple B models. These are all advantages over using a visualisation that has to be written within the B models specification.

The following chapters explain how the input file formats were chosen and which restrictions they underlay. Additionally, there will be a small example to understand the functionality of those files. After that, the implementation of the current architecture is described.

3.2.1 VisB file

The VisB file is the key element that makes a VisB visualisation possible. For that, a file format is needed that can be converted into a Java object. At best it would have a built-in Java library to convert its data into the needed Java objects and is already used in ProB2-UI.

JSON [6] is a file format that most Java developers are familiar with. The JSON format "is a lightweight data-interchange format. It is easy for humans to read and write [...] [and] easy for machines to parse and generate" [15]. Additionally, the JSON format is already used in ProB2-UI. This makes the visualisation specification file easy to maintain as well as easy to use.

The visualisation file, called VisB file, consists of a simple structure, in which there are visualisation items and visualisation events declared. These can be used to manipulate the SVG image in the user interface later on. Listing 1 shows an example of the structure of a VisB file.

```
{
  "svg":"button.svg",
  "items":[
    {
      "id":"button",
      "attr":"fill",
      "value":"IF button=TRUE THEN \"green\" ELSE \"red\" END"
    }
  ],
  "events":[
    {
      "id":"button",
      "event":"press_button"
    }
  ]
}
```

Listing 1: Minimal Example for VisB file

As one can see, a VisB item contains an id, an attribute and a value. The value contains a B formula with return values that can be evaluated to JavaScript values of SVG attributes. How the evaluation of this works, is explained in subsection 3.3.

To be able to click on the SVG items, an event has to be specified. The reason why the visualisation file is separated into items and events is that only SVG elements with corresponding events in the JSON file get an on-click functionality when loading the files. The thought behind this is to keep the interaction between Java and JavaScript as clean as possible.

Every VisB event contains an id, an event and the event's predicates if they are necessary. Parameters for events have to be expressed as predicates. An example for

this is shown in section 5. The JSON keyword is *predicates*, though, because these events can contain other predicates in addition to the parameters. The event refers to an operation in the B model.

The VisB file also contains the path to the SVG file. This path can be relative to the VisB file, or absolute.

Further examples of the structure of a VisB file can be found in section 5.

3.2.2 SVG file

SVG, or Scalable Vector Graphics, is an XML-based markup language. The language is designed to describe two-dimensional images. SVG is a widely used image format that can be displayed in all modern browsers and created with a text editor [10].

Because SVG is a markup language, the user can easily change the image's attributes in a text editor. There are no further programs needed to write an SVG file. Though it is suggested the user uses a program to construct larger SVG images because it is easier than writing the SVG specifications oneself. For that reason, using SVG as the image format for the visualisation plugin is a set requirement from the beginning, as mentioned in section 1.

An SVG file has a wide variety of elements which can be modified with attributes. Those elements have an id attribute. With this attribute, an SVG element can be modified in other attributes. To further explain the functionality, one has to take a look at the structure of an SVG file. You can see an example in Listing 2.

```
1 <svg height="200" width="200">
2   <circle id="button" cx="100" cy="100" r="80"
3     stroke="black" stroke-width="3" fill="green" />
4 </svg>
```

Listing 2: Minimal Example for SVG file

The SVG file must contain elements which have corresponding ids to those ids used in the VisB file. If one takes a look at line 5 and 12 in Listing 1, for example, "button" has to be the id of an SVG element in the SVG file. In this example, "button" is also the id of the circle element in Listing 2.

In short, the SVG image is a set choice from the beginning because it is very user-friendly, and can be modified easily.

3.3 Implementation of VisB

In the implementation of VisB, the focus is to achieve a maintainable tool. To achieve this goal, the architecture is structured as simple as possible. In addition to that, the code is well documented.

The current architecture, as shown in Figure 4 contains the VisB-UI and the VisB Java Core. In this section, only the VisB Java Core is explained. The section additionally describes the interaction between VisB Java Core and VisB-UI. The VisB-UI layout is inspected in more detail in section 4, where the layout itself is evaluated.

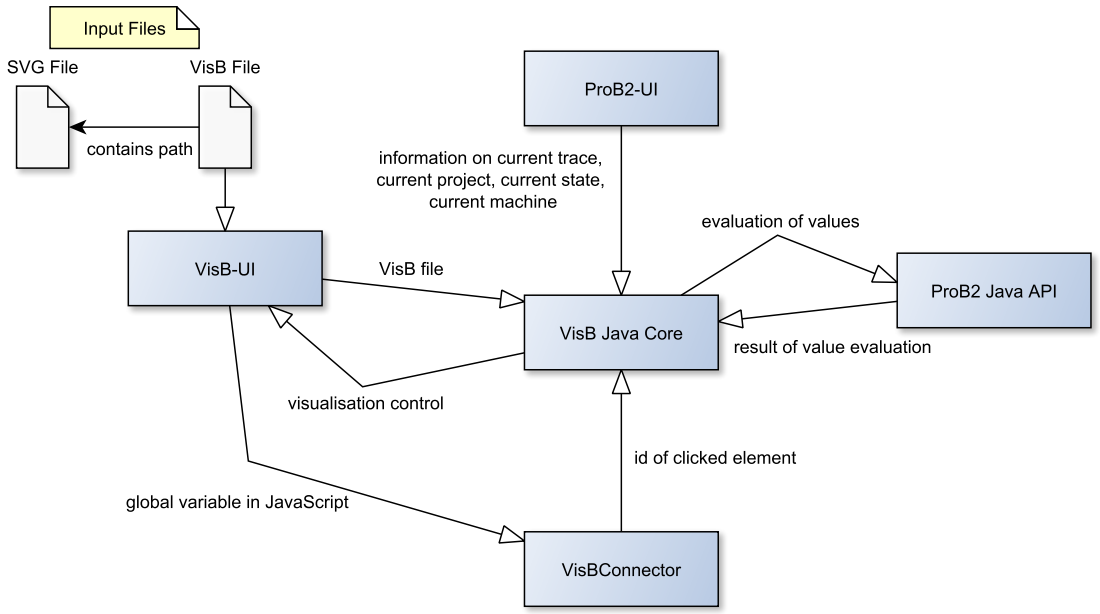


Figure 4: Current Internal Structure

Figure 4 shows that the VisB Java Core gets information from ProB2-UI about the current project. This information is needed for changes in the VisB-UI as well as starting and stopping a visualisation automatically after a VisB file is loaded.

The VisB Java Core interacts with the ProB2 Java API to evaluate the B formulas, which are written in the values of an item in the VisB file. These B formulas return strings that represent SVG attribute values, which are used in a jQuery. This jQuery is created in the VisB Java Core as well. It is then sent to the VisB-UI to manipulate the visualisation. This is, what is meant by "visualisation control" in Figure 4.

The user selects the VisB file, which contains the path to the SVG image used for this visualisation. The first approach is to make this SVG image path absolute. By taking a relative path, this path could either be relative to the ProB2-UI project file, the file of the B model or the VisB file and this could confuse the user. However, because collaborative repositories are often used in IT development, the SVG image file can be relative to the VisB file, or absolute. Moreover, it is clarified that the SVG image file path is relative to the VisB file path. After the VisB file is selected, the VisB-UI sends it to the VisB Java Core to prepare it for visualisation, as shown in Figure 4.

In subsection 3.2 it is mentioned that the JSON format comes with a built-in Java library to convert the content of the file to Java objects. Those Java objects are part

of the VisB Java Core. They have to be implemented manually to correspond with the chosen JSON layout. E.g. a VisBVisualisation object holds the path to the SVG file, the visualisation items, and the visualisation events. This is an internal representation of the JSON file.

The current state of the loaded B model affects the visualisation. Information about the current state is taken from ProB2-UI, as shown in Figure 4. Whenever the current state of the model changes after the initialisation state, the VisB-UI adapts corresponding to the user input in the VisB file. The very simple B model seen in Listing 15 describes a button that can be pressed one time only.

This model has two states, one in which the button is not pressed and one in which the button is pressed. The button is represented by a boolean value. In ProB2, a model's specification can be visualised as a state space graph. In the state space graph, a node represents a specific state and the path from the root state to a specific state represents a trace in the state space. After the current state changes, the values of the VisB items are newly evaluated.

A very simple VisB file can be seen in Listing 1. In its context, the button is represented by a circle. By inspecting the contents of the SVG image in Listing 2, one can see that without manipulation this circle is set as green. However, after initialising the visualisation with the B model in Listing 15 the button is red. This can be seen in Figure 6

To achieve this functionality, the VisB Java Core directs the evaluation of the VisB item values, as shown in Figure 4. For that, each B expression in those item values is evaluated on the current state.

In ProB2, IF and LET can be used as expressions and predicates, though this functionality is not ordinary for B interpreters [22]. This makes it possible to construct the B expression, which returns a string value, as shown in Listing 1.

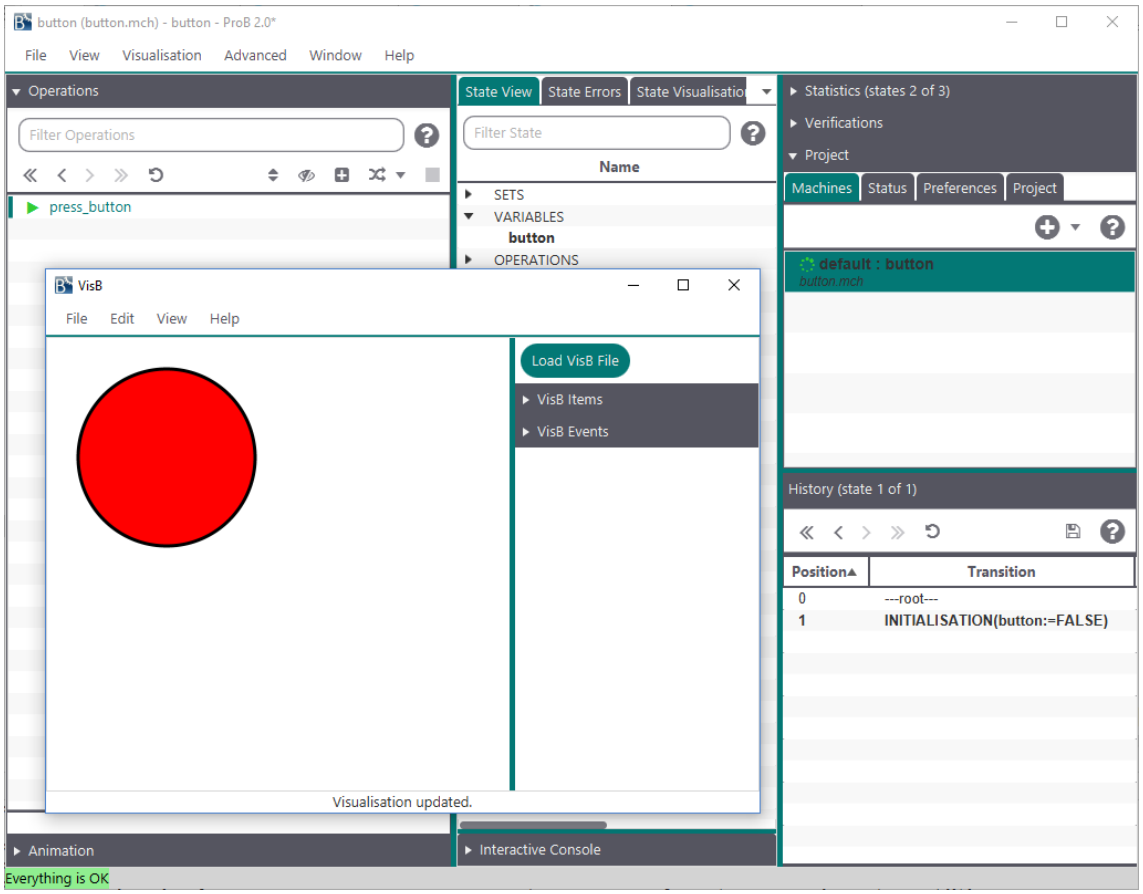
Those B expressions are then evaluated one by one by the B Interpreter of the ProB Java API. After that, they are translated into a jQuery library call within the VisB Java Core. A jQuery library call manipulates the SVG image in the VisB UI, which can be read upon in the next chapter. When the jQuery library call is executed, the SVG image changes accordingly to its new values for its attributes. Because the B expressions are currently evaluated one by one, the performance could be drastically improved in future work, as described in section 8.

The VisB Java Core sends the previously mentioned jQuery library calls to VisB-UI, as mentioned before. Those jQuery library calls all have the same structure. The structure is shown in Listing 3.

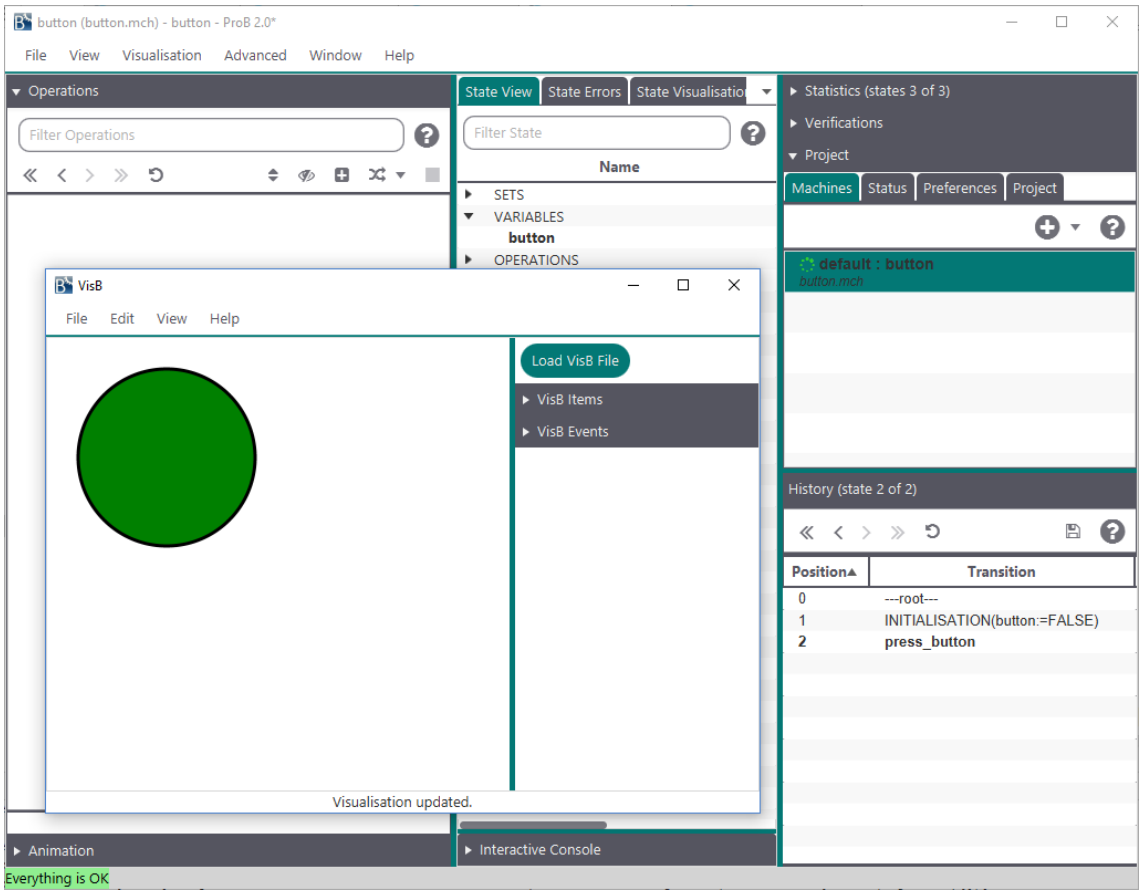
```
1 changeAttribute("#button", "fill", "red");
```

Listing 3: jQuery call for minimal example

ChangeAttribute is a JavaScript function, which is hardcoded into the HTML embedding of the SVG image. Because of that, it is possible to use the ids of the SVG



(a) After Initialisation



(b) Button is Pressed

Figure 6: Visualisation of the Button

elements to manipulate the SVG.

Most of the string values are tested before they are used in the jQuery library call. The testing of the right usage of values for the corresponding attributes helps the user to find mistakes and errors in their files. However, not all of those values can be tested, without a JavaScript interpreter.

The connection between the VisB UI and the JavaScript interaction is implemented within the VisB Java Core. In Figure 4, it is represented by the VisBConnector. This connector makes it possible to click on an SVG element and execute an operation of a B model in ProB2-UI.

In this particular example, the circle in the SVG file in Listing 2 with the id "button" can be clicked because there is a visualisation event in the VisB file. This event describes a B operation. For this particular event, no predicates are needed.

If the SVG element is clicked, the VisBConnector notifies the VisB Java Core with the id of the SVG element. After that, the corresponding event for this id is found and executed.

The VisB Java Core uses the current trace to try and execute the operation. The information about the current trace is provided by the ProB2-UI, as one can see in Figure 4. If the operation can be executed, the resulting trace is set as the new current trace. If an operation cannot be executed, the user is notified over the information label in the VisB-UI. If an operation needs parameters, the procedure is done with the corresponding predicates.

Momentarily, there can only be one operation for one SVG element, meaning there can be only one VisB event for one id. That way, developers have a better overview of which element is clicked and which operation has to be executed. The functionality of clicking an SVG element and automatically run more than one operation is discussed in section 8.

4 VisB-UI Layout

In the development of the VisB-UI layout, one has to consider a different goal than in the development of the internal structure. While the user usually does not interact with the internal structure of a tool, the user interacts with the user interface. That means, the layout of the interface inevitably determines the usability of the tool. Therefore, making the layout of VisB-UI as simple as possible for the user is one of the goals for this thesis.

Additionally, another goal is set: Leaving or creating a place for additional features or improving the plugin’s extensibility. Additional features are not necessary for building simple visualisations, which is the goal of this thesis. However, it would be beneficial for the user experience to implement them in the future. It should be possible to implement new features as seamlessly as possible. Therefore, the layout should hold the capability of adding additional features to it.

In the following sections, it is described how the layout of VisB-UI is developed. Furthermore, it is explained, how the functionality and structure of a layout determine the usability. In the end, the current VisB-UI layout is described and its advantages and disadvantages are explained.

4.1 Development of the VisB-UI Layout

In the early stages of the user interface, the idea is to display the SVG image in the VisB-UI in a window itself. That means the controls would be in a different window than the SVG image. The reason for this layout idea is, that it is very easy to understand and intuitive to control. An illustration of it can be seen in the Figure 7.

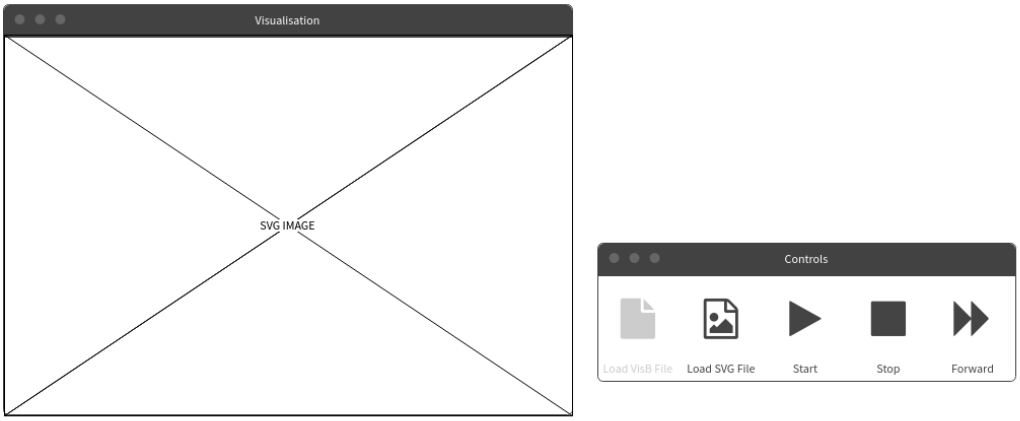


Figure 7: Separate Windows

The advantages of this idea are the possibility to have the visualisation view in one window and the controls in another. This would be beneficiary, e.g. for beamer presentations. As well as, the possibility to automatically animate the visualisation, which is further explained in section 8.

Then again, this layout does make it possible to edit and construct a visualisation later on in development. In Figure 7 one can see, that controls for starting and stopping a visualisation would have been necessary. However, with the current implementation, the visualisation cannot be started or stopped by the user. That is because the visualisation is getting updated, whenever the model updates. Additionally, this layout lacks user information, such as a loading bar or an information label.

In conclusion, the idea of separate windows is interesting and will be used for future work, but the overall structure of this layout is not compatible with the current internal structure and the possibility to add additional features in the future.

This next idea is the first layout of the VisB-UI. In this layout, one could find the

possibility to load the VisB file and SVG file separately as well as the SVG image and the visual representation of the VisB items in the same window, as shown in Figure 8.

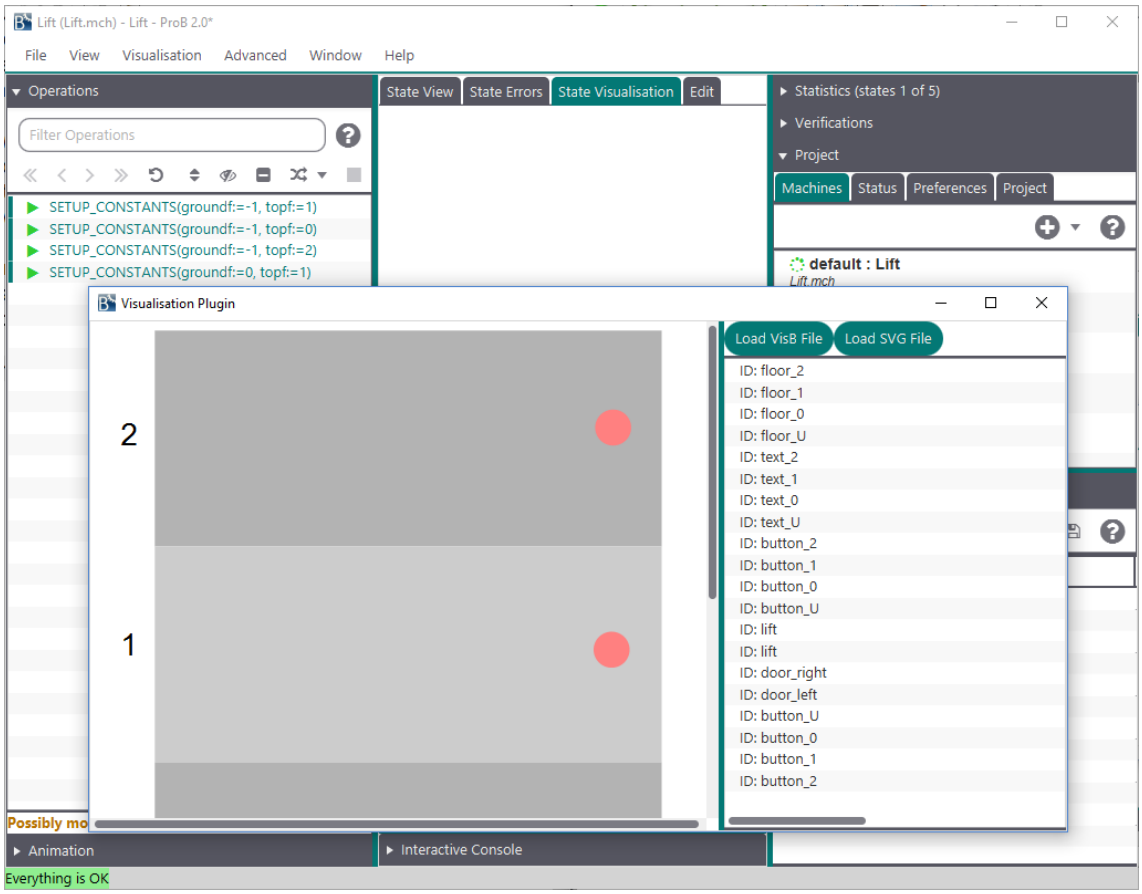


Figure 8: Early Stage of VisB-UI Layout

The screenshot shows an early stage of the current UI layout with a visualisation of the Lift loaded, which is further explained in subsection 5.1.

As shown in Figure 8, there are two buttons, which allow the user to load the VisB file and SVG file manually in the stage. In addition to that, there is the possibility to see the visualisation items. They are represented as strings in a list and the VisB events are not displayed at all. This is because of the idea, that one probably did not want to edit the VisB events over the UI in the future. In this idea, the SVG image could be independently loaded into the UI.

On the one hand, this layout could be easily added onto. New features could be in the same window or get a separate window entirely. Because the layout itself is simple enough to add on features very seamlessly. That is necessary because the user should not have to relearn how to use the tools functions after updates for new features.

However, the layout is not user-friendly enough, yet. There is no information or feedback for the user. The two buttons for loading the SVG image and the VisB file are

intricate to deal with. Two files have to be added manually over the stage before a visualisation is possible. Lastly, the way the VisB items are represented in Figure 8 is not professional and the VisB events are not displayed at all. This is bad for development if an editing possibility is implemented in the future.

In summary, this layout idea is a relatively good choice for additional features, because it leaves a lot of room for future additions, but it does not visually fit to ProB2-UI and lacks in usability. That is why, in the next section, this simple layout is made more user-friendly with new features. The layout basis is the one seen in Figure 8.

4.2 Current VisB-UI Layout

The following section explains the final changes to the VisB-UI in consideration to the last chapter. Additionally, it highlights the advantages and disadvantages of this layout.

After testing and analysing the layout in Figure 8, the decision is made to add a menu bar. This helps the user to navigate in the UI. The menu bar also holds the key for more additional features, which can be accessed over menu items.

Two of those additional features are implemented in that menu bar. A help menu item would lead to a short user manual. The zooming menu items are added to the menu bar. The possibility to resize or scale the SVG image is something, that highly improves the functionality and usability of this Plugin.

Another aspect that is missing in the two layout versions before is feedback. In ProB2-UI, there is a status bar which displays information for the user. Possible pieces of information from ProB2-UI are, whether the model is loaded correctly if an error occurred somewhere or even the information that a file is processed. Because this feature is so useful, and improves the usability, it is added in the form of a status label in the VisB-UI layout. The plain label is chosen because a coloured status bar would draw the users attention from the actual visualisation.

The amount of effort a user has to make to start a working visualisation is decreased by adding the path of the SVG image to the VisB file. This can be done because it is not planned for the user to be able to edit the SVG image itself within VisB.

These changes mentioned above can be seen in Figure 9.

The screenshot shows the current layout of VisB-UI. As one can see, it is colour-coordinated with ProB2-UI. Those colours are chosen because the ProB2-UI already has distinct colours, which the user connects with ProB2. Additionally, to make the user experience as seamless as possible, the same styles for buttons and labels were chosen.

As one can see, there are now four menu items, which help the user interact with VisB. The button for loading the VisB file is kept in the UI and is now used to add the VisB file, which contains the SVG path, as shown in section 5.

Last but not least, the VisB items and VisB events are displayed in a more professional

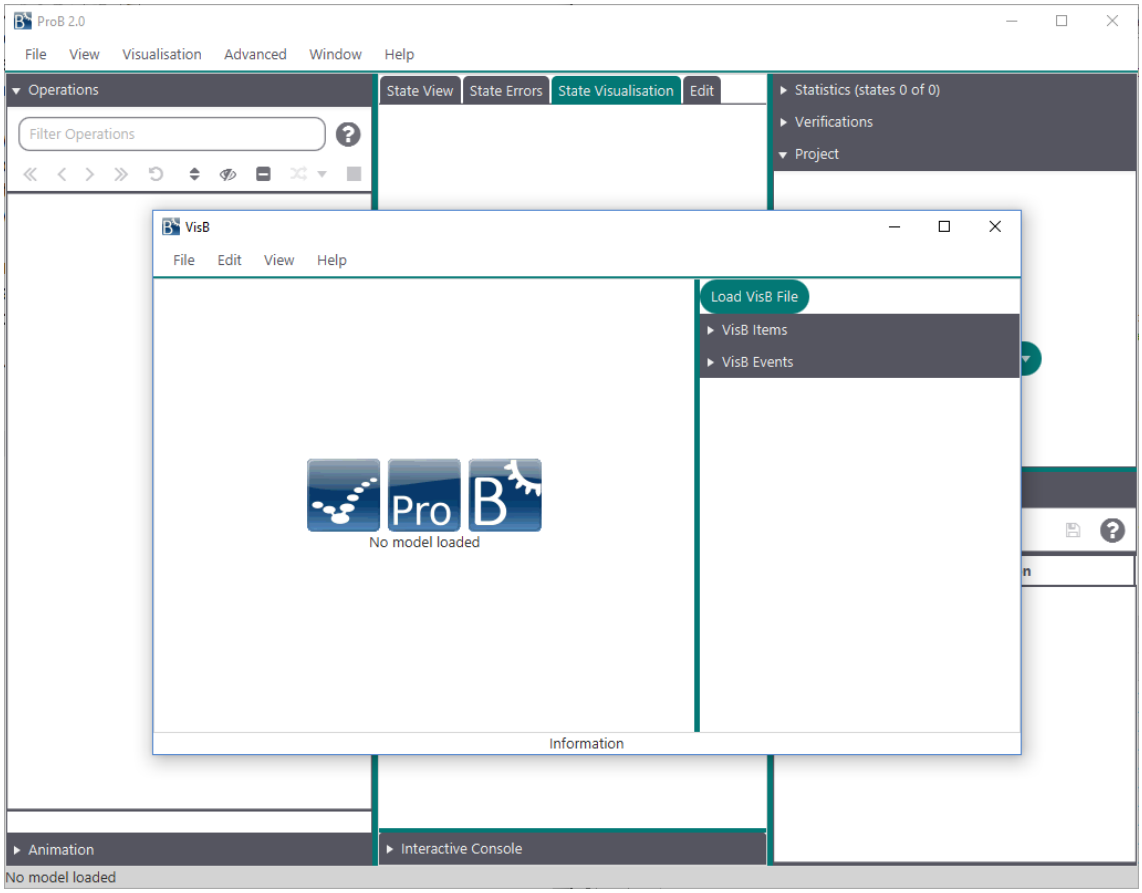


Figure 9: Current VisB-UI Layout on Top of ProB2-UI

way, which matches with ProB2-UI. Now each list has its view. This makes it easier to add the feature of editing items and events later on.

There is a small disadvantage to this layout in comparison to the previous ones, though. The disadvantages are that the layout does not make it possible for the user to stop the visualisation without closing the window.

To sum this up, the layout of the VisB-UI currently has a lot of usable features. These enable the user to easily interact with VisB-UI and switch seamlessly between it and ProB2-UI. Additionally, the usability of the layout has been improved from previous stages. Lastly, additional features can easily be implemented with the current layout. Therefore, the goals that were set for the VisB-UI layout have been accomplished with the layout shown in Figure 9.

5 Visualisation Examples

The following uses two B models to illustrate VisB. One is the model of a lift, the other one is a variation of the eight queens puzzle [9]. The B models for the following visualisation examples can be found in the ProB Public Examples [4]. In this chapter, these two examples are visualised with VisB.

5.1 Lift

The B model of this example is shown in Listing 16. It specifies a simple lift, that can move up and down on four or less different floors. These floors are restricted by a top floor and a ground floor, which are set as integer constants.

The lift is represented by three variables: the current floor is an integer value between the ground floor and the top floor, the variable for direction can be either up or down and the last variable states if the doors are opened or not.

The environment of the lift, on the other hand, is represented by the top floor, the ground floor and eight buttons, four on the inside and four on the outside of the lift.

Moreover, the specification contains an invariant to ensure that the ground floor is smaller than the top floor at all times.

However, the lift needs the possibilities to move up, move down, change the direction as well as opening and closing the doors. The user also needs to be able to press the buttons, that previously were added to the environment of the lift. Both are realised with operations in B.

For this B model, one can construct an SVG image, that shows how the current state of the lift looks like after initialisation of the B model. What the SVG image for this example looks like without manipulation can be seen in Figure 10.

After creating an image with the desired components one has to set ids. Each element that one intends to manipulate in the visualisation needs an id. Note that the SVG

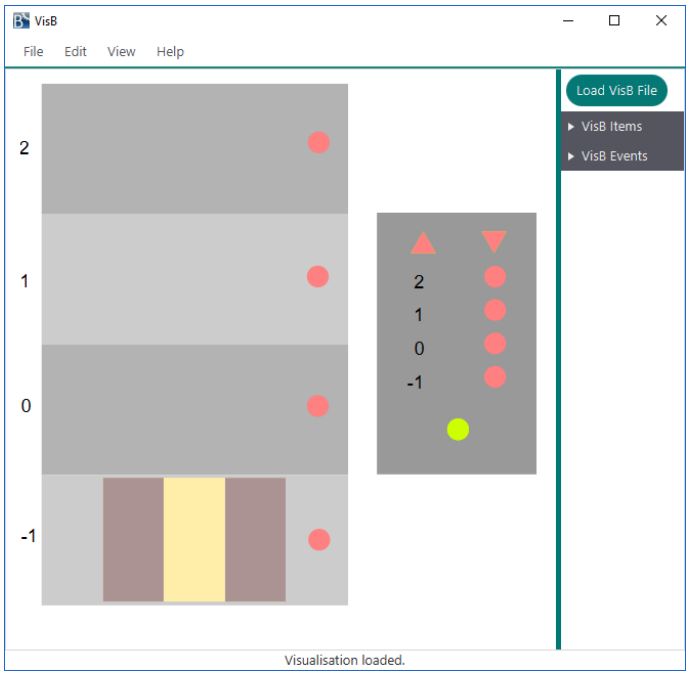


Figure 10: SVG Image for the Lift Visualisation

image in Figure 10 contains overlapping items. This will be used later on to use more than one event in one place.

The SVG image for the visualisation is created and the identifiers for the SVG elements are set. Now one can think about what each element of the SVG image does when the B model is running. This information will be used to create the VisB file. For this visualisation, the ground floor and top floor can vary. What one can do now, is to hide unused floors for the current visualisation. In the created SVG image one floor consists of five SVG elements. The corresponding VisB items can be seen in Listing 4. This is how the VisB file would look like if one did not use the possibility to group the elements.

```
...
    {
      "id":"floor_2",
      "attr":"visibility",
      "value":"IF topf=2 THEN \"visible\" ELSE \"hidden\" END"
    },
...
    {
      "id": "text_2",
      "attr": "visibility",
      "value": "IF topf=2 THEN \"visible\" ELSE \"hidden\" END"
    },
```

```
...
    {
      "id": "inside_text_2",
      "attr": "visibility",
      "value": "IF topf=2 THEN \"visible\" ELSE \"hidden\" END"
    },
...
    {
      "id": "button_2",
      "attr": "visibility",
      "value": "IF topf=2 THEN \"visible\" ELSE \"hidden\" END"
    },
...
    {
      "id": "inside_2",
      "attr": "visibility",
      "value": "IF topf=2 THEN \"visible\" ELSE \"hidden\" END"
    },
...
```

Listing 4: Change "visibility" Attribute

By grouping the elements, one can change some of the attributes of each of the elements in this group with one VisB item.

Now that the SVG elements are grouped, the VisB file only contains one item for visibility, as shown in Listing 5. This approach changes the visibility of the five elements with one VisB item.

```
...
    {
      "id":"gFloor_2",
      "attr":"visibility",
      "value":"IF topf=2 THEN \"visible\" ELSE \"hidden\" END"
    },
...
```

Listing 5: The Benefits of Grouping SVG Elements

For this visualisation, the user needs more than visible floors, though. The lift is represented by three rectangles. Two of those represent the doors and the other one represents the inside of the lift. This visualisation tricks the user by changing the colour of the inside of the lift to make it appear as if the doors are closed. Additionally, every button that can be pushed should change colour. The SVG attribute needed to change the colour of an SVG element is called "fill", it can be seen in Listing 6.

```
...
    {
      "id":"lift",
      "attr":"fill",
      "value":"IF door_open=TRUE THEN \"#ffeeaa\" ELSE \"#ac9393\" END"
    },
...
    {
      "id":"button_U",
      "attr":"fill",
      "value":"IF -1:call_buttons THEN \"#FF0000\" ELSE \"#FF8080\" END"
    },
...
    {
      "id":"inside_U",
      "attr":"fill",
      "value":"IF -1:inside_buttons THEN \"#FF0000\" ELSE \"#FF8080\"
         END"
    },
...
```

Listing 6: Change "fill" Attribute

One can observe that the symbol " which is used around the return values has to be escaped with the symbol \. This is because in JSON " is a special character, that expects a string value, similarly to Java. This is something the user has to keep in mind when writing the B formulas for each item.

Additionally, the lift has a VisB item for changing its position in the SVG image. One cannot change the x or y values of groups, which is why these values have to be changed for each part that moves. An example for this attribute change can be seen in Listing 7. One can work around that by nesting SVG images, as it can be seen in subsection 5.2.

```
...
    {
      "id":"lift",
      "attr":"y",
      "value":"IF cur_floor=2 THEN \"grounf+3\" ELSIF cur_floor=1 THEN \
         "76.974\" ELSIF cur_floor=0 THEN \"150.474\" ELSE \"224.574\"
         END"
    },
    {
      "id":"door_right",
      "attr":"y",
      "value":"IF cur_floor=2 THEN \"3.207\" ELSIF cur_floor=1 THEN \"
```

```
          76.974\" ELSIF cur_floor=0 THEN \"150.474\" ELSE \"224.574\"
          END"
    },
    {
      "id":"door_left",
      "attr":"y",
      "value":"IF cur_floor=2 THEN \"3.207\" ELSIF cur_floor=1 THEN \"
          76.974\" ELSIF cur_floor=0 THEN \"150.474\" ELSE \"224.574\"
          END"
    },
...
```

Listing 7: Change "y" Attribute

The overlapping items, mentioned before are meant to simplify the usages of reversing the lift, moving the lift and opening and closing the doors. The idea is to hide the item which's operation cannot be executed and show the item that can be executed on the current trace and, therefore, has a on-click functionality. Because, it is enough to show and hide the item that is on top of the other one in the SVG image, the VisB items shown in Listing 8 are enough to achieve this behaviour. This is similarly done for reversing and moving the lift up and down.

```
...
    {
      "id":"open_door",
      "attr":"visibility",
      "value":"IF door_open=TRUE THEN \"hidden\" ELSE \"visible\" END"
    },
...
    {
      "id":"close_door",
      "event":"close_door"
    },
    {
      "id":"open_door",
      "event":"open_door"
    },
...
```

Listing 8: Hiding Not Executable Elements

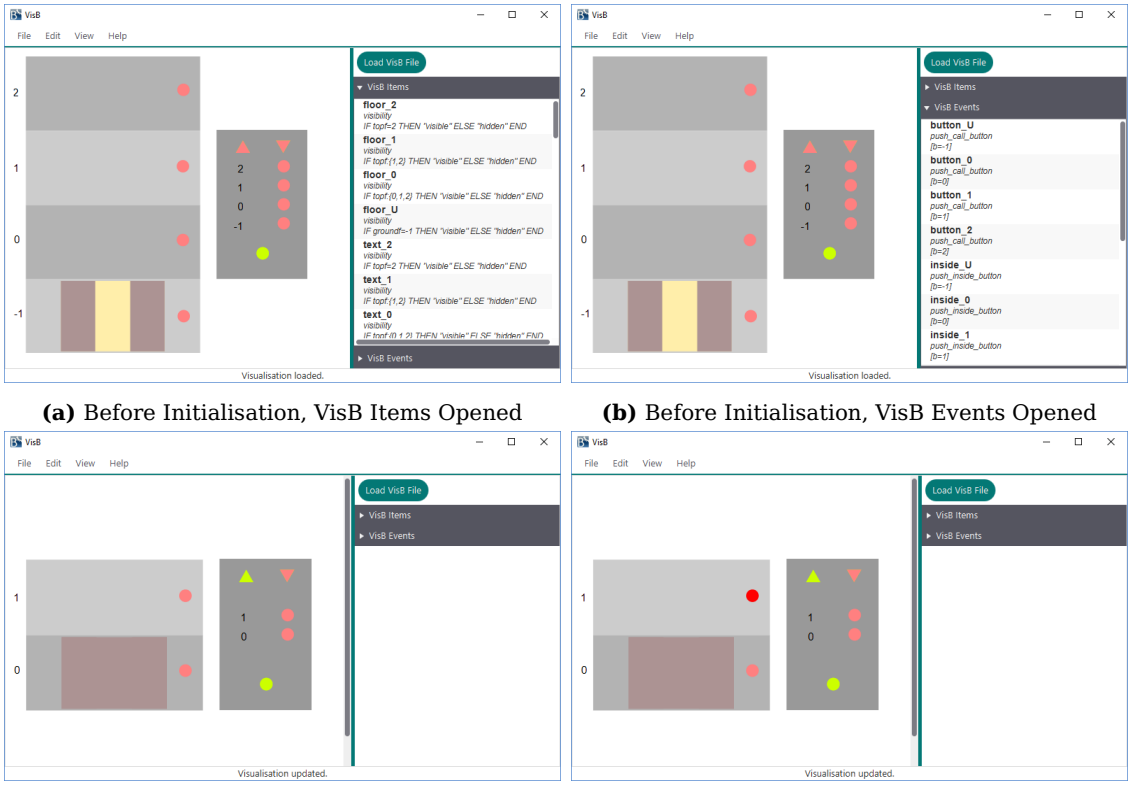
To execute an operation with a parameter, that parameter has to be set as an additional predicate for the executable event. An example of that can be seen in Listing 9.

```
...
    {
      "id":"button_U",
```

```
      "event":"push_call_button",
      "predicates":[
        "b=-1"
      ]
    },
...
```

Listing 9: Event with Parameters

After all of the buttons have their events, similar to the one shown in Listing 9 the visualisation for the lift is finished. The complete VisB file that is created in this visualisation example can be found in Listing 18. A small example of how two of the states look like in VisB can be found in Figure 12. A bigger example of a whole use case can be found in Appendix C.



(a) Before Initialisation, VisB Items Opened

(b) Before Initialisation, VisB Events Opened

(c) After Initialisation, topf=1, groundf=0

(d) After call_button 1 is pressed

Figure 12: Example of two States of the Lift Visualisation

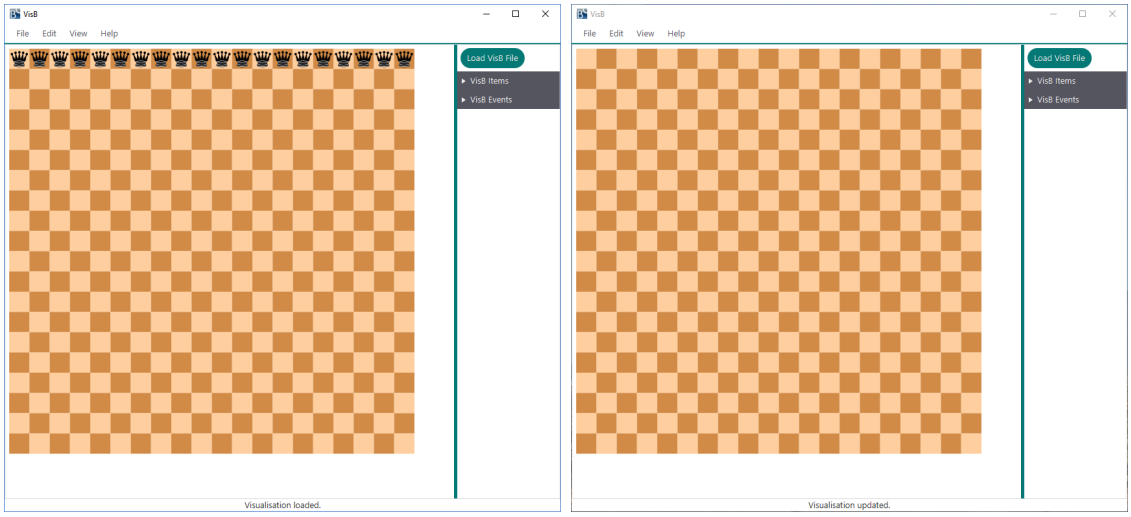
5.2 N-Queens

The following logic puzzle is a more general version of the *eight queens puzzle* [9]. The problem of this puzzle is to place eight queens on an 8×8 chessboard so that no two queens do attack each other. Now the n-queens puzzle is the same, but with n queens on an $n \times n$ chessboard.

The specification of the B model is defined as follows. It is listed in Listing 17. In the beginning, the user can set n. This n is then used to generate the $n \times n$ chessboard. This can be realised with setting up constants and the initialisation of the B model.

After the initialisation, the user can set the queens. For setting the queens, the operation `SetQueens(i,j)` is used. Additionally to that, if a queen of a certain column is already set, the user can change that queen's column. This can be done with the `ChangeQueen(i,j)` operation.

For this visualisation, there are already SVG images that can be used to create the chessboard seen in Figure 14. These images can be found in the ProB Public Examples [4]. To produce the image seen in Figure 14 these SVG images were nested and combined to a bigger SVG file.



(a) Before Initialisation, 20×20 **(b)** After Initialisation, 20×20

Figure 14: N-Queens SVG Image for Visualisation

The simplest and fastest approach of the visualisation uses $n + n \times n$ SVG items n nested SVG images, that represent the queens and $n \times n$ tiles, that represent the chessboard.

A tile is visible if its row or its column is smaller than `N`. `N` is a constant set by the user.

However, when continuing this approach for the chessboard, it would lead to a similar problem one had with the lift visualisation. In the comparison of Listing 10 and Listing 11 this problem becomes clear.

```
...
    {
      "id": "tile8x1",
      "attr": "visibility",
      "value" : "IF 8<=n & 1<=n  THEN \"visible\" ELSE \"hidden\" END"
    },
...
    {
      "id": "tile8x2",
      "attr": "visibility",
      "value" : "IF 8<=n & 2<=n  THEN \"visible\" ELSE \"hidden\" END"
    },
...
    {
      "id": "tile8x3",
      "attr": "visibility",
      "value" : "IF 8<=n & 3<=n  THEN \"visible\" ELSE \"hidden\" END"
    },
...
```

Listing 10: Visibility of Tiles without Grouping

If each tile has a VisB item, the number of VisB items needed for an $n \times n$ chessboard is clearly $n \times n$. However, the chessboard is always of the form $n \times n$, this is why this approach is simplified by the process of grouping the items so that the amount of VisB items needed becomes n.

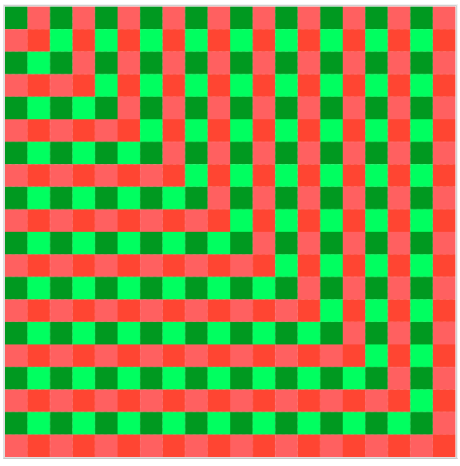


Figure 15: N-Queens Grouping

For this approach, the SVG elements representing the tiles are grouped as visualised in Figure 15. The tile (i, j) is in group k if $i \leq k$ and $j \leq k$, but only if $i = k$ or $j = k$.

```
...
    {
      "id": "gTiles8",
      "attr": "visibility",
      "value" : "IF 8<=n  THEN \"visible\" ELSE \"hidden\" END"
    },
...
```

Listing 11: Visibility of Tiles with Grouping

The SVG elements representing the queens are not grouped. Each queen is invisible after initialisation. They become visible if the queen's column is set in the model. This can be checked with the B predicate `q1:dom(queens)`, as shown in Listing 12.

```
...
    {
      "id": "svgQueen1",
      "attr": "visibility",
      "value" : "IF 1:dom(queens) THEN \"visible\" ELSE \"hidden\" END"
    },
    {
      "id": "svgQueen1",
      "attr": "y",
      "value" :"IF  1|->2:queens THEN \"45\" ELSIF 1|->3:queens THEN \"
         90\" ELSIF 1|->4:queens THEN \"135\" ELSIF 1|->5:queens THEN \
         "180\" ELSIF 1|->6:queens THEN \"225\" ELSIF 1|->7:queens THEN
          \"270\" ELSIF 1|->8:queens THEN \"315\" ELSIF 1|->9:queens
         THEN \"360\" ELSIF 1|->10:queens THEN \"405\" ELSIF 1|->11:
         queens THEN \"450\" ELSIF 1|->12:queens THEN \"495\" ELSIF
         1|->13:queens THEN \"540\" ELSIF 1|->14:queens THEN \"585\"
         ELSIF 1|->15:queens THEN \"630\" ELSIF 1|->16:queens THEN \"
         675\" ELSIF 1|->17:queens THEN \"720\" ELSIF 1|->18:queens
         THEN \"765\" ELSIF 1|->19:queens THEN \"810\" ELSIF 1|->20:
         queens THEN \"855\" ELSE \"0\" END"
    },
    {
      "id": "svgQueen1",
      "attr": "fill",
      "value" : "IF is_attacked(1) & 1:dom(queens) THEN \"red\" ELSE \"
         black\" END"
    },
...
```

Listing 12: VisB Items for Queens

The VisB item used to set the y coordinate of the queen can be seen in Listing 12 in line 10. The B formula used to set the y coordinate is constructed as follows. There is only one queen for each column. The exact position of the queen in column i can be found out by checking for each row j if `i|->j:queens` is true. The exact position for the y coordinate is $45 * (j - 1)$ for this example because the height of the images is 45 pixels. E.g. the queen's y coordinate on position $j = 1$ is $y = 0$, for $j = 2$ it's $y = 45$, etc.

The last VisB item determines the colour of the queen. That is achieved by using the SVG attribute "fill". The "fill" of the queen is "red" whenever the queen is attacked and it is a set queen, otherwise the queen's "fill" is "black".

The specification of the model gives the user the possibility to set and change the queens on the chessboard. It is not possible in VisB to have multiple operations for one SVG element, but one can work around that by redefining the user input in a new operation, as seen in Listing 13.

It is, additionally, possible to use machine inclusion and it would probably be more elegant. For this, another model has to be specified that extends the model seen in Listing 17 with a new operation.

However, this example redefines the user input since it is not important how the new operation is implemented for the visualisation of it. Moreover, changing the code is easier to understand at this point. The VisB item shown in Listing 14 achieves the functionality of setting and changing queens by clicking on the tiles of the chessboard.

```
1  TryQueen(i,j) = PRE i:1..n & j:1..n THEN
2      IF i /: dom(queens) THEN
3          SELECT i:1..n & j:1..n & i /: dom(queens) THEN
4              queens(i) := j
5          END ELSE
6          SELECT i:1..n & j:1..n & i : dom(queens) & j /= queens(i) THEN
7              queens(i) := j
8          END
9       END
10    END;
```

Listing 13: Redefining the User Interaction for the Visualisation

```
1  ...
2      {
3        "id": "tile1x1",
4        "event": "TryQueen",
5        "predicates" : ["i=1","j=1"]
6      },
7      {
8        "id": "tile2x1",
9        "event": "TryQueen",
```

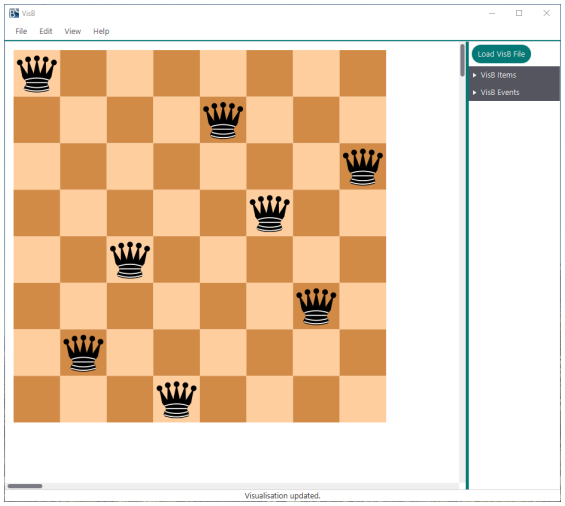
```
10      "predicates" : ["i=2","j=1"]
11    },
12 ...
```

Listing 14: VisB Events for Tiles

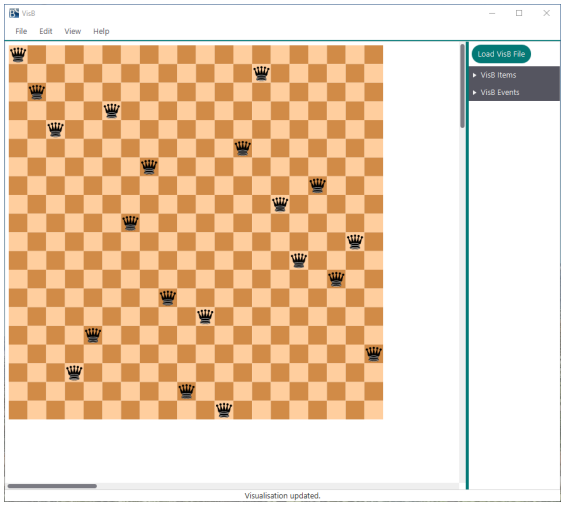
Listing 14 shows that one VisB Event for each tile (i, j) is set. With that, VisB executes the operation `TryQueen(i, j)` for each tile (i, j).

With the visualisation specification set, one can now visualise the n-queens problem for an $n \times n$ chessboard. A full use case for a 10×10 chessboard can be seen in Appendix C. Figure 17 shows possible solutions for $n = 8$, $n = 20$, $n = 120$.

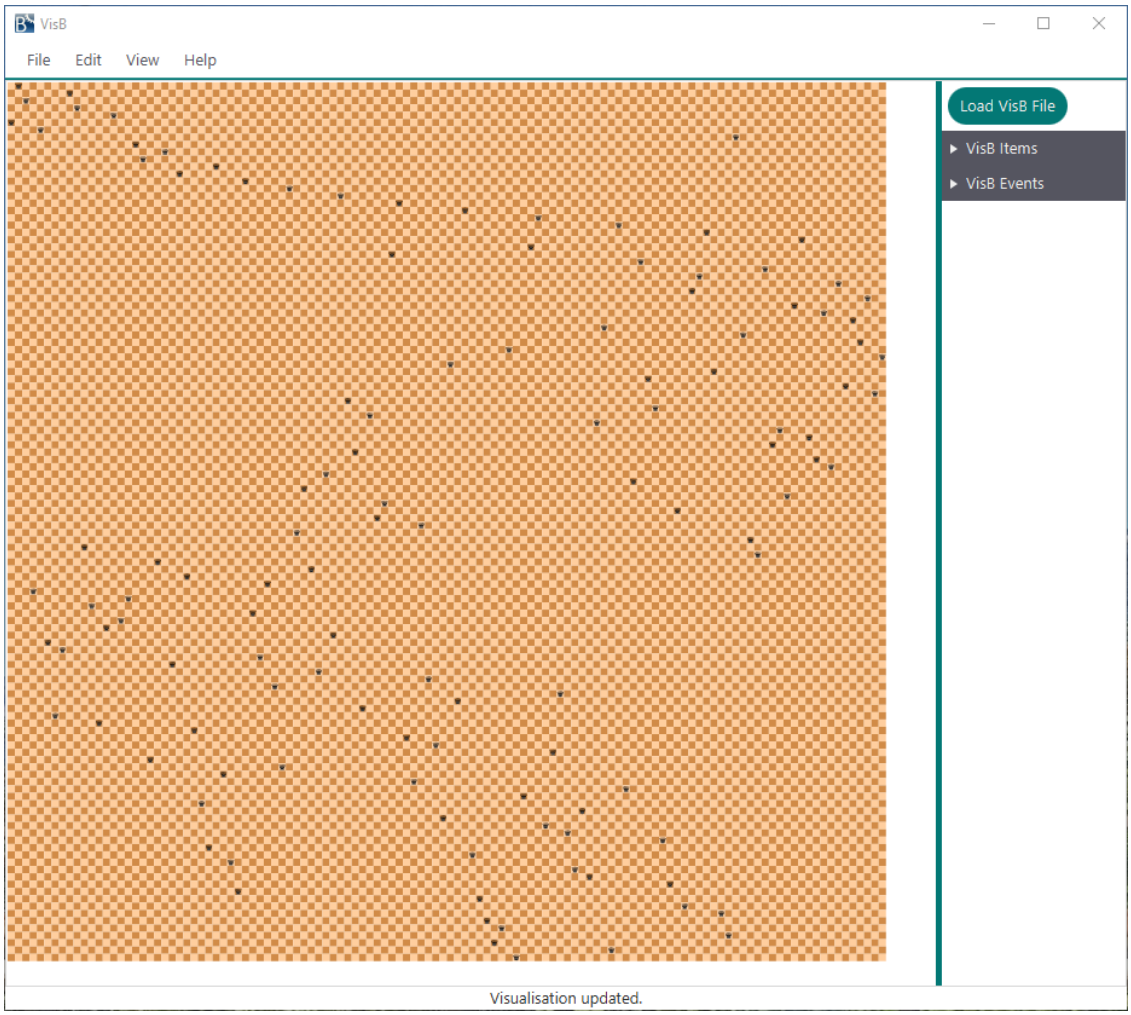
The case $n = 120$ is an example that cannot be visualised with the previous constructed VisB items because they only have y coordinated for 20 tiles. It is used, however, to evidence the possibility to visualise bigger state spaces in VisB. The construction of a VisB file for $n = 120$ is done similarly to the discussed construction for $n = 20$.



(a) Solution for $n = 8$



(b) Solution for $n = 20$



(c) Solution for $n = 120$

Figure 17: N-Queens Visualisation Solutions

6 Related Work

Visualisation and animation have always been very important for the ProB team. Over the years of development, there have been several visualisations for ProB2.

Currently, two of the following visualisation tools can be maintained and are usable in ProB2-UI. One of these tools is the *ANIMATION_FUNCTION* [21], which was one of the first visualisation tools built for ProB. The JavaFX based *visualisation mechanism* [13], created by Heinzen can be used in ProB2-UI as well. *B-Motion Studio* [18] for ProB2 is a visualisation tool that was created by Ladenberger. This visualisation, however, is currently not usable in the current version of ProB2-UI.

In the following chapters, each of the visualisation tools is summarised. After that, the visualisations are analysed in consideration of the goals set in this thesis and finally compared to VisB. However, these tools were created for different reasons, which is explained in these summaries as well.

6.1 Animation Function

One of the first visualisation tools for ProB was introduced in a paper that Leuschel and Bendisposto published in 2007 [5]. The motivation for this animation tool is to create a visual prototype for a B model. The motivation is that domain experts and formal method experts can communicate properties of the B model more easily.

However, this tool, based on Macromedia Flash, is inconvenient for students and lecturers, as the setup and the development of glueing code are too difficult [21].

The *ANIMATION_FUNCTION* was developed in 2008 [21]. The motivation was to create a visualisation tool which is easier to use. The authors explain that the *ANIMATION_FUNCTION* has to be declared in the DEFINITIONS. This tool is implemented in B. Paths to images have to be declared in those as well. With this tool, the user can visualise B models in a grid layout by using the B language. An example of the *ANIMATION_FUNCTION* written for the n-queens problem can be seen in Listing 19.

The advantages of the *ANIMATION_FUNCTION* are the following: The tool has been added to ProB2-UI and is very useful for simple visualisations. The *ANIMATION_FUNCTION* needs a small amount of maintenance because it is very sophisticated. Another positive aspect is, that the user writes in the B language only.

On disadvantages are that the functionality of the tool is rather restricted by the grid layout, which is inconvenient for more complex visualisations. Additionally, the tool requires writing the *ANIMATION_FUNCTION* which "can still be a considerable challenge" [19].

In conclusion, the *ANIMATION_FUNCTION* is still a very useful, sophisticated tool for creating visualisations and animations of B models. In comparison, the *ANIMATION_FUNCTION* is not as user-friendly as VisB, because it requires a rather complicated function written in B. This is not as intuitive as selecting the value for an

attribute for an SVG element. However, VisB is less sophisticated than the *ANIMATION_FUNCTION* and therefore lacks in features the *ANIMATION_FUNCTION* has, e.g. right click mouse menus for selecting an operation. VisB is also more flexible in creating visualisations because the *ANIMATION_FUNCTION* is bound by the a grid layout.

6.2 BMotion Studio and BMotionWeb

BMotion Studio was introduced in 2009, in a paper by Leuschel, Bendisposto and Ladenberger and was created to simplify the process of creating a visualisation. They explained, that the *ANIMATION_FUNCTION* is still too complicated to use [19]. BMotion Studio uses controls and observers. The controls are graphical representations of important aspects of the model, while the observers are used to "link controls to the model's state" [19].

However, *BMotion Studio* does not have features for validation. In 2016 Ladenberger, therefore, presents *BMotionWeb*, a "novel graphical environment" created to validate "interactive safety-critical systems by creating interactive formal prototypes" [18]. "BMotionWeb is the successor of BMotion Studio", as stated by Ladenberger in [18]. What Ladenberger means is that this new visualisation approach makes it possible to validate models by creating visual prototypes.

BMotionWeb is a complex tool for creating formal prototypes, that includes a great number of features, such as a graphical environment for creating visualisations and a scripting language for more complex prototypes [18]. The tool is based on web-technologies like JavaScript and HTML and works with SVG, just like VisB does [18]. Additionally, it similarly is created with a *visualisation template* in JSON format. One also can add a Groovy Script for further modifications or write JavaScript code for the visualisation.

VisB is mainly focused on usability, maintainability, and the possibility to create a wide range of B models. *BMotionWeb*, on the other hand, is focused on more complex goals, such as prototyping and validation. *BMotionWeb* is written mostly in JavaScript and Java, though a great amount of the code is JavaScript [18]. Because *BMotionWeb* is focused on different aspects and is mostly written in JavaScript, the code is hard to understand for other members of the ProB team. Therefore, this tool is not maintainable at the moment.

All in all, this tool is user-friendly, because one can create a visualisation in a small amount of time. *BMotionWeb* additionally has the capability of creating a visualisation for a wide range of B models because it supports nondeterministic behaviour which VisB is currently not able to do. However, the tool is not maintainable in the long-term because its architecture is very complex. When comparing this tool to VisB, it has advantages in the sense that it has a very wide range of B models that can be visualised with it, but it has its disadvantage in terms of maintainability. Both tools are user-friendly and work with modern web technologies, though using *BMotionWeb* requires more knowledge about those technologies, especially when one wants to use

JavaScript or Groovy additions.

6.3 Visualisation Mechanism

Heinzen introduces the *visualisation mechanism* for ProB2-UI in his master's thesis. He states in its motivation that a ProB user needs basic knowledge of SVG, HTML and JavaScript for a simple visualisation with BMotionWeb [13]. The *visualisation mechanism* is written in Java and JavaFX. It requires, theoretically, one JavaFX class to be written for a visualisation to be possible, but it is hard to achieve a useful visualisation with only one JavaFX class with that tool.

The Java class that represents the visualisation can only access built-in Java objects from the JavaFX library, such as rectangles, circles, and other simple shapes, e.g. in the lift visualisation example [14]. If one wants to create a more complex visualisation the user has to write more JavaFX classes, as one can see in the bridges visualisation example in Heinzen's thesis [14]. Unfortunately, there are not many tools, that provide a user-friendly way of creating a more complex image in JavaFX.

However, there are tools that translate an SVG image to JavaFX, though they are limited to the SVGPath [24] class and therefore do not have all of the specifications of SVG images, as mentioned in subsection 3.1.

The *visualisation mechanism* is optimised in performance and multiple machines can be visualised with one visualisation [13]. Additionally, it needed an in-memory compiler for the Java visualisations, with user-friendly error messages [13].

The code of the *visualisation mechanism* is maintainable and the *visualisation mechanism* can be used for standalone visualisations for B models. However, because the visualisations are rather specialised on one domain, there is still a lot of code for every visualisation to maintain, which makes it overall harder to maintain. Especially if the usage of code for the visualisation mechanism changes.

In short, the *visualisation mechanism* is not user-friendly, because the user has to write complicated JavaFX classes to create a visualisation. Additionally, images for visualisation are very intricate to construct. In comparison, VisB is more user-friendly and visualisations can be constructed more easily. The *visualisation mechanism* can be maintained well, but the visualisations themselves are much harder to maintain. VisB relies on a file system so that major changes internally less likely affect already created visualisations. That means VisB has an advantage in maintenance.

7 Conclusion

All in all, the goals of this thesis were accomplished during this work. In brief, these goals were to build a visualisation plugin, VisB, for ProB2-UI that is maintainable, user-friendly, and can visualise a broad range of B models.

It is investigated how the architecture of VisB is developed. There are different approaches for a visualisation plugin: Creating it in JavaFX only, by using the SVGPath class and on the other hand, implementing a web application fully based on JavaScript. Both of these approaches were shown to be unpractical for they do not accomplish the goals of this thesis. The final approach of using a combination of JavaFX and JavaScript, however, was discovered to fulfill every goal.

The thesis shows that VisB is maintainable because its architecture is easy to understand and well documented. In VisB's architecture, the logic and the user interface are isolated from each other while interacting with each other. This enhances the maintainability of the plugin.

Furthermore, the VisB files are easy to read and easy to work within the development. The usage of JSON is useful because it comes with the advantages of built-in libraries in Java. SVG has a wide variety of elements and functionalities and was, therefore, a good choice for this plugin. These file formats further enhance the maintainability of the tool because the logic and the user interface of the visualisation itself are separated.

Similarly, it is investigated whether the user interface of VisB is user-friendly. Operating the tool can be done instinctively because only a few things have to be understood when using the tool. The additional goal, to find a layout suitable for additional features, was accomplished as well. This is accomplished because the user will not have to relearn the user interface, after developing features and adding them to the extensive layout in the future.

Moreover, this thesis achieves the goal to create a tool that enables the user to create simple visualisations for B models. This is shown in two different examples, that was easily created and visualised in VisB. For both examples, there are use cases that show how domain experts can use the visualisation created by formal methods experts.

In the end, this thesis compares VisB to previous visualisation tools. Each previously discussed tool has its advantages and disadvantages. However, VisB is the only tool that accomplishes all three goals that are set in this thesis. VisB is maintainable, user-friendly, and a broad range of visualisations for different kinds of B models can be achieved with it.

8 Future Work

With all this, the work is not done, yet. There are numerous ideas on how to improve VisB further, while still containing the goals of this thesis. In the future, the following ideas could be further developed and researched. However, the intention of this thesis should be kept in mind, while further developing this tool.

1. **Implementing an editor for manipulating VisB files over VisB-UI.** As mentioned before in this thesis, the usability would increase, when one could implement an editor for VisB files. When working with VisB at the moment, the B user has to write a large amount of code, e.g. for repeating items. This could be simplified by an editor, e.g. the editor could repeat certain code fragments with different ids.

2. **Investigating and increasing the performance of VisB.** Though this is not seen as necessary for this thesis because it has a different focus, one could increase the performance of VisB. This could be done by finding another solution for the evaluation of values.

3. **Standalone visualisation with VisB.** In *BMotionWeb* [18] standalone visualisations were possible. That means creating visualisations that can be executed separately from ProB2-UI. However, further research has to be done on how to implement such functionality in VisB and if it would be beneficial.

4. **Possibility to separate visualisation from editor layout.** If the editor is developed, one should find a way to separate the visualisation from the editor. Currently, the goal was to get an extensive visualisation layout. This extensiveness should be used, however, to make it possible to separate the views. This should be done because it would be beneficial e.g. for presentations and lectures, where the VisB code should not be seen.

5. **The integration of VisB into ProB2-UI.** In the long run, a goal for this project should be to integrate VisB into ProB2-UI because the tool can be very useful for users of ProB2-UI.

A List of B Models used in this Thesis

```
MACHINE button
VARIABLES
  button
INVARIANT
  button:BOOL
INITIALISATION
  button := FALSE
OPERATIONS
  press_button = PRE button=FALSE THEN
    button:=TRUE
  END
END
```

Listing 15: Minimal Example for B model

```
MACHINE Lift
/* A simple model of a lift without a controller; The controller
   will be added in a refinement (by refining the lift moving operations
      ) or
   by a CSP controller */
DEFINITIONS
  FLOORS == (groundf .. topf);
  all_buttons_pressed == (inside_buttons \/ call_buttons);
  /* Note: one could make a slightly simpler spec by only keeping a
     single all_buttons_pressed variable */
  ASSERT_LTL == "G ([push_call_button(groundf)] -> X F {cur_floor=
     groundf & door_open=TRUE})";
  ASSERT_LTL2 == "G ([push_inside_button(topf)] -> X F {cur_floor=topf &
      door_open=TRUE})"
  /* These LTL properties are violated: e.g., the machine can reverse
     infinitely without moving */
CONSTANTS groundf,topf
PROPERTIES
 groundf:INT & topf:INT & groundf < topf
VARIABLES cur_floor, inside_buttons, door_open, call_buttons,
   direction_up
INVARIANT
 cur_floor : FLOORS &
 inside_buttons <: FLOORS &
 door_open : BOOL &
 call_buttons <: FLOORS &
 direction_up : BOOL
INITIALISATION
```

```
23   cur_floor := groundf || inside_buttons := {} || door_open := FALSE ||
        call_buttons := {} || direction_up := TRUE
24 OPERATIONS
25   /* The lift operations : */
26   move_up = PRE door_open = FALSE & cur_floor <topf & direction_up=TRUE
        THEN
27     cur_floor := cur_floor +1
28   END;
29   move_down = PRE door_open = FALSE & cur_floor>groundf & direction_up=
        FALSE THEN
30     cur_floor := cur_floor - 1
31   END;
32   reverse_lift_down = PRE direction_up=TRUE THEN direction_up := FALSE
        END;
33   reverse_lift_up = PRE direction_up=FALSE THEN direction_up := TRUE END
        ;
34   open_door = PRE door_open = FALSE & cur_floor : all_buttons_pressed
        THEN
35     door_open := TRUE
36   END;
37   close_door = PRE door_open = TRUE THEN
38     door_open := FALSE ||
39      /* clear requests as floor now visited: */
40     inside_buttons := inside_buttons - {cur_floor} || call_buttons :=
          call_buttons - {cur_floor}
41   END;
42
43   /* The user interface : */
44   push_inside_button(b) = PRE b:FLOORS & b/: inside_buttons & b/=
        cur_floor THEN
45     inside_buttons := inside_buttons \/ {b}
46   END;
47   push_call_button(b) = PRE b:FLOORS & b/: call_buttons THEN
48     call_buttons := call_buttons \/ {b}
49   END
50 END
```

Listing 16: Lift Specifications in B from the ProB Public Examples [4]

```
MACHINE QueensWithEvents
CONSTANTS n
PROPERTIES
 n : NATURAL &
 n < 20
DEFINITIONS
      SET_PREF_TIME_OUT == 6000;
      SET_PREF_CLPFD == TRUE;
      SET_PREF_MAX_INITIALISATIONS == 20;
      SET_PREF_MAX_OPERATIONS == 10;
      MAX_OPERATIONS_SetQueen == 1000;
      MAX_OPERATIONS_ChangeQueen == 1000;
      MAX_OPERATIONS_SolveAny == 4;

      is_attacked(q1) == q1:dom(queens) &
                         #q2.(q2:dom(queens) & q2 /= q1 &
                              (no_attack(q1,q2,queens) => queens(q1) =
                                 queens(q2)));
      pos_is_attacked(q1,q1row) ==
                         #q2.(q2:dom(queens) & q2 /= q1 &
                              (no_attack_pos(q1,q1row,q2,queens(q2)) =>
                                 q1row = queens(q2)));
      no_attack(q1,q2,board) == no_attack_pos(q1,board(q1),q2,board(q2))
          ;
      no_attack_pos(q1,q1row,q2,q2row) == (q1row+q2-q1 /= q2row & q1row-
          q2+q1 /= q2row);
      Solution(board) == (
         board :  perm(1..n) /* for each column the row in which the
             queen is in */
         &
         !(q1,q2).(q1:1..n & q2:2..n & q2>q1 => no_attack(q1,q2,board) )
        )
VARIABLES queens
INVARIANT
  queens : (1..n) +-> (1..n)
INITIALISATION
  queens := {}
OPERATIONS
  Solve = ANY solution WHERE
      Solution(solution) &
      !x.(x:dom(queens) => solution(x)=queens(x))
  THEN
     queens := solution
  END;
```

```
40   SolveFuzzy = ANY solution WHERE
41       Solution(solution) &
42       !x.(x:dom(queens) => solution(x):{queens(x)-1,queens(x),queens(x)
             +1})
43   THEN
44      queens := solution
45   END;
46   SetQueen(i,j) = SELECT i:1..n & j:1..n & i /: dom(queens) THEN
47      queens(i) := j
48   END;
49   ChangeQueen(i,j) = SELECT i:1..n & j:1..n & i : dom(queens) & j /=
         queens(i) THEN
50      queens(i) := j
51   END;
52   r<--Get(yy) = PRE yy:dom(queens) THEN r:= queens(yy) END
53 END
```

Listing 17: N-Queens Problem in B from the ProB Public Examples without Animation Function [4]

B Complete VisB File Examples

```
{
  "svg":"lift_groups.svg",
  "items":[
    {
      "id":"gFloor_2",
      "attr":"visibility",
      "value":"IF topf=2 THEN \"visible\" ELSE \"hidden\" END"
    },
    {
      "id":"gFloor_1",
      "attr":"visibility",
      "value":"IF topf:{1,2} THEN \"visible\" ELSE \"hidden\" END"
    },
    {
      "id":"gFloor_0",
      "attr":"visibility",
      "value":"IF topf:{0,1,2} THEN \"visible\" ELSE \"hidden\" END"
    },
    {
      "id":"gFloor_U",
      "attr":"visibility",
      "value":"IF groundf=-1 THEN \"visible\" ELSE \"hidden\" END"
    },
    {
      "id":"lift",
      "attr":"fill",
      "value":"IF door_open=TRUE THEN \"#ffeeaa\" ELSE \"#ac9393\" END"
    },
    {
      "id":"lift",
      "attr":"y",
      "value":"IF cur_floor=2 THEN \"3.207\" ELSIF cur_floor=1 THEN \"
         76.974\" ELSIF cur_floor=0 THEN \"150.474\" ELSE \"224.574\"
         END"
    },
    {
      "id":"door_right",
      "attr":"y",
      "value":"IF cur_floor=2 THEN \"3.207\" ELSIF cur_floor=1 THEN \"
         76.974\" ELSIF cur_floor=0 THEN \"150.474\" ELSE \"224.574\"
         END"
    },
```

```
    {
      "id":"door_left",
      "attr":"y",
      "value":"IF cur_floor=2 THEN \"3.207\" ELSIF cur_floor=1 THEN \"
         76.974\" ELSIF cur_floor=0 THEN \"150.474\" ELSE \"224.574\"
         END"
    },
    {
      "id":"button_U",
      "attr":"fill",
      "value":"IF -1:call_buttons THEN \"#FF0000\" ELSE \"#FF8080\" END"
    },
    {
      "id":"button_0",
      "attr":"fill",
      "value":"IF 0:call_buttons THEN \"#FF0000\" ELSE \"#FF8080\" END"
    },
    {
      "id":"button_1",
      "attr":"fill",
      "value":"IF 1:call_buttons THEN \"#FF0000\" ELSE \"#FF8080\" END"
    },
    {
      "id":"button_2",
      "attr":"fill",
      "value":"IF 2:call_buttons THEN \"#FF0000\" ELSE \"#FF8080\" END"
    },
    {
      "id":"inside_U",
      "attr":"fill",
      "value":"IF -1:inside_buttons THEN \"#FF0000\" ELSE \"#FF8080\"
         END"
    },
    {
      "id":"inside_0",
      "attr":"fill",
      "value":"IF 0:inside_buttons THEN \"#FF0000\" ELSE \"#FF8080\" END
         "
    },
    {
      "id":"inside_1",
      "attr":"fill",
      "value":"IF 1:inside_buttons THEN \"#FF0000\" ELSE \"#FF8080\" END
         "
```

```
    },
    {
      "id":"inside_2",
      "attr":"fill",
      "value":"IF 2:inside_buttons THEN \"#FF0000\" ELSE \"#FF8080\" END
         "
    },
    {
      "id":"reverse_up",
      "attr":"visibility",
      "value":"IF direction_up=TRUE THEN \"hidden\" ELSE \"visible\" END
         "
    },
    {
      "id":"reverse_down",
      "attr":"visibility",
      "value":"IF direction_up=FALSE THEN \"hidden\" ELSE \"visible\"
         END"
    },
    {
      "id":"open_door",
      "attr":"visibility",
      "value":"IF door_open=TRUE THEN \"hidden\" ELSE \"visible\" END"
    }
  ],
  "events":[
    {
      "id":"button_U",
      "event":"push_call_button",
      "predicates":[
        "b=-1"
      ]
    },
    {
      "id":"button_0",
      "event":"push_call_button",
      "predicates":[
        "b=0"
      ]
    },
    {
      "id":"button_1",
      "event":"push_call_button",
      "predicates":[
```

```
        "b=1"
      ]
    },
    {
      "id":"button_2",
      "event":"push_call_button",
      "predicates":[
        "b=2"
      ]
    },
    {
      "id":"inside_U",
      "event":"push_inside_button",
      "predicates":[
        "b=-1"
      ]
    },
    {
      "id":"inside_0",
      "event":"push_inside_button",
      "predicates":[
        "b=0"
      ]
    },
    {
      "id":"inside_1",
      "event":"push_inside_button",
      "predicates":[
        "b=1"
      ]
    },
    {
      "id":"inside_2",
      "event":"push_inside_button",
      "predicates":[
        "b=2"
      ]
    },
    {
      "id":"close_door",
      "event":"close_door"
    },
    {
      "id":"open_door",
```

```
      "event":"open_door"
    },
    {
      "id":"up",
      "event":"move_up"
    },
    {
      "id":"down",
      "event":"move_down"
    },
    {
      "id":"reverse_up",
      "event":"reverse_lift_up"
    },
    {
      "id":"reverse_down",
      "event":"reverse_lift_down"
    }
  ]
}
```

Listing 18: Complete VisB File for Lift Model

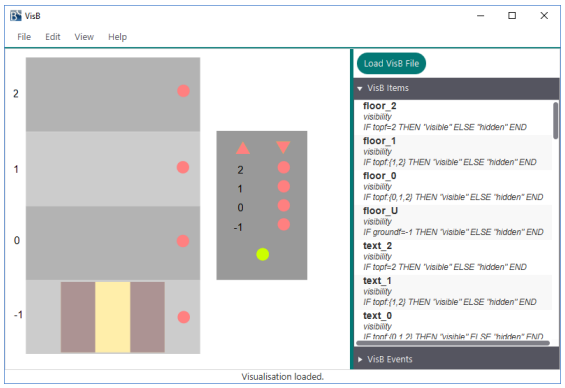
C Additional Examples

C.1 Use Case for Lift Visualisation

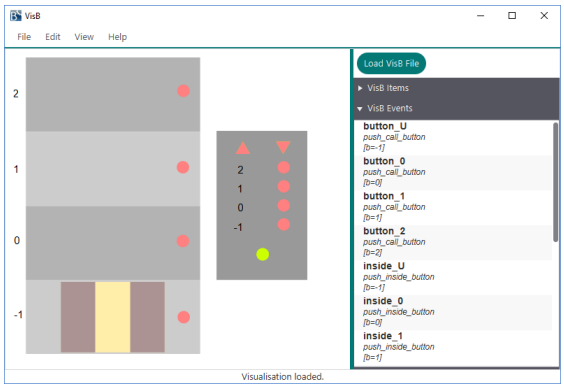
In the use case example in Figure 19 and Figure 21 the lift starts at the ground floor. The lift picks up guests from the ground floor and transports them to the top floor. After that, the lift goes back into the state after the initialisation.

C.2 Use Case for N-Queens Visualisation

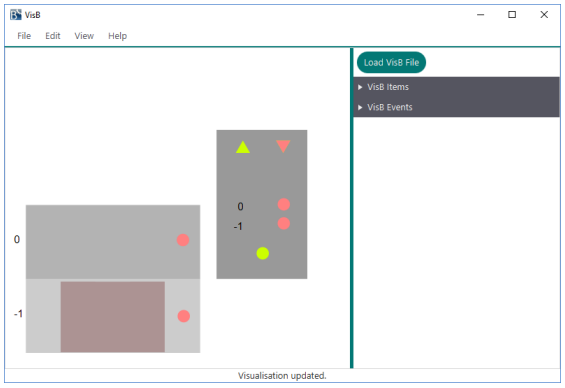
In the following images, a domain expert is able to find a solution for the n-queens problem, with $n = 10$.



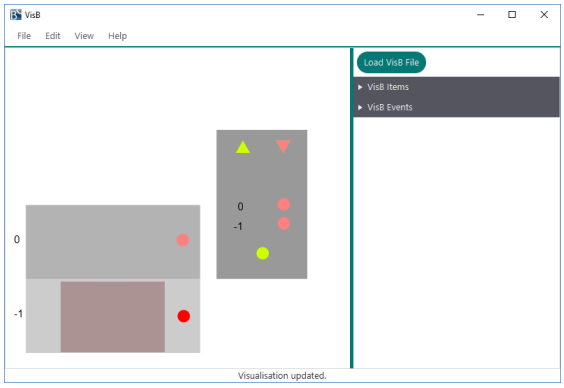
(a) Before Initialisation, VisB Items Opened



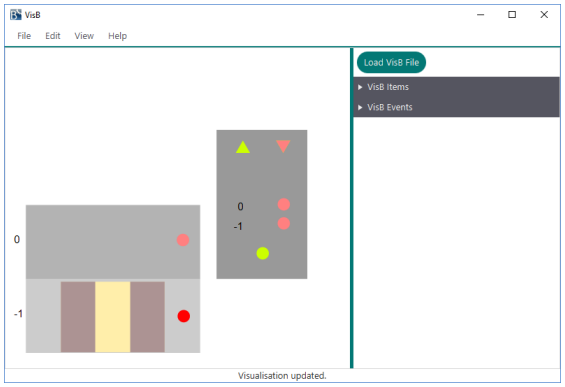
(b) Before Initialisation, VisB Events Opened



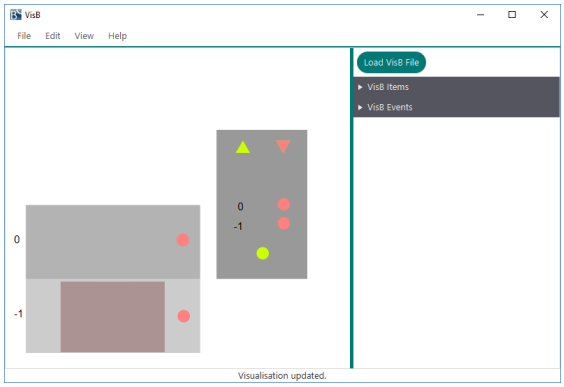
(c) After Initialisation, topf=0, groundf=-1



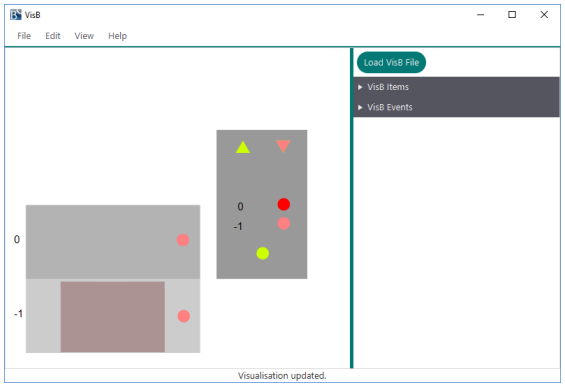
(d) A guest presses the button on ground floor.



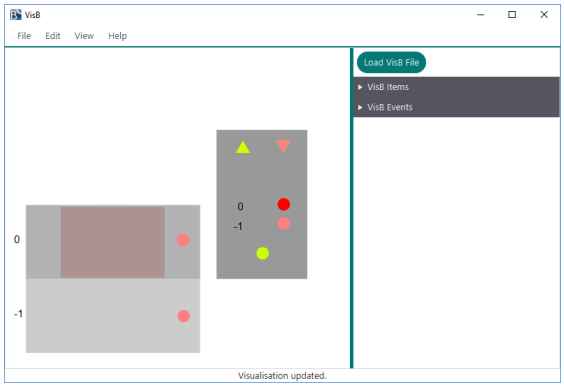
(e) The doors open. The guests move inside of the lift.



(f) The doors close again.



(g) The inside button for the top floor is pressed.



(h) The lift moves up.

Figure 19: Use Case Example for Lift Visualisation Part 1

(a) The doors open. The guests leave the lift.

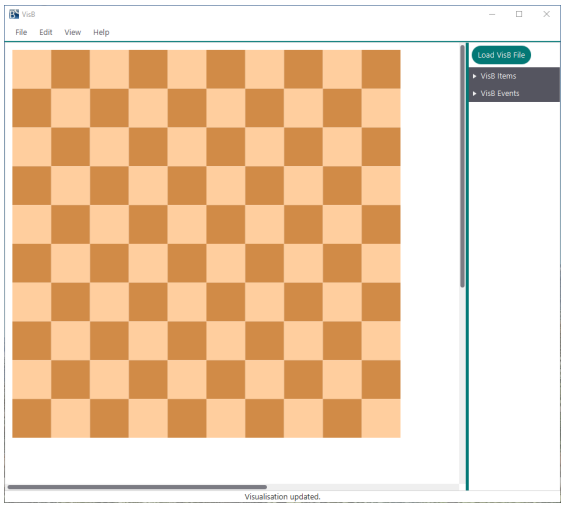
(b) The doors close.

(c) The lift reverses down.

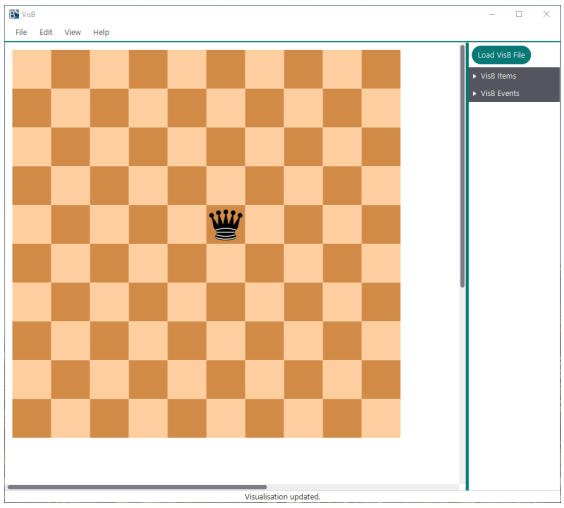
(d) The lift moves down.

(e) The lift reverses up. The same state as in Figure 19 (c) is reached.

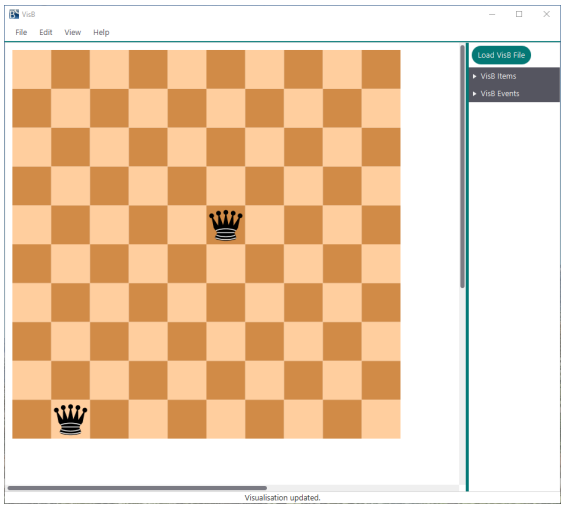
Figure 21: Use Case Example for Lift Visualisation Part 2



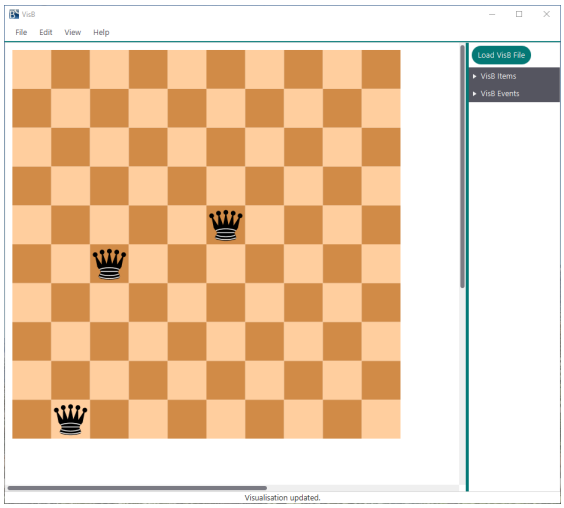
(a) After Initialisation, $n = 10$



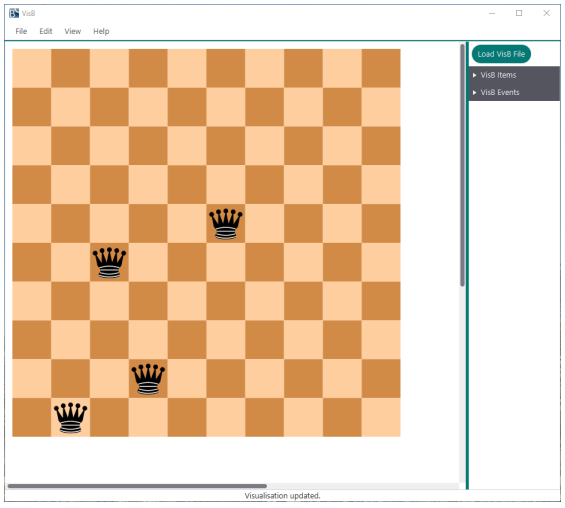
(b) Set the queen in column 6 to row 5.



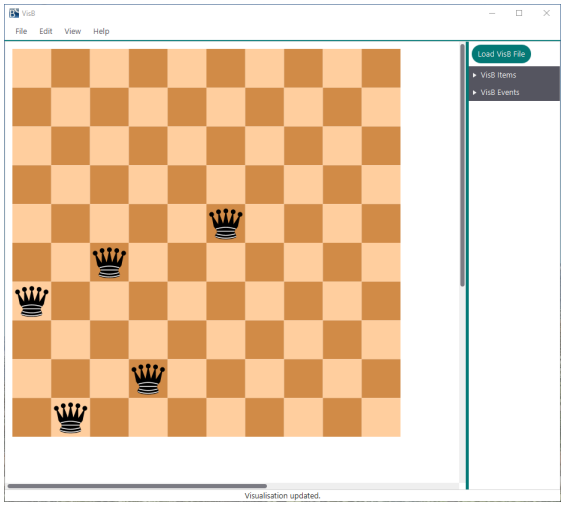
(c) Set the queen in column 2 to row 10.



(d) Set the queen in column 3 to row 6.

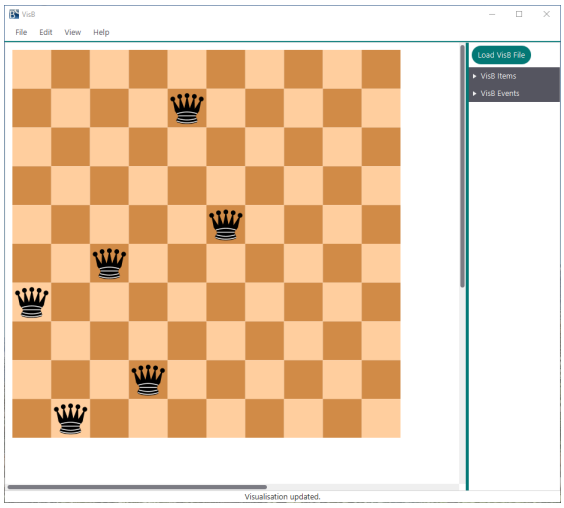


(e) Set the queen in column 4 to row 9.

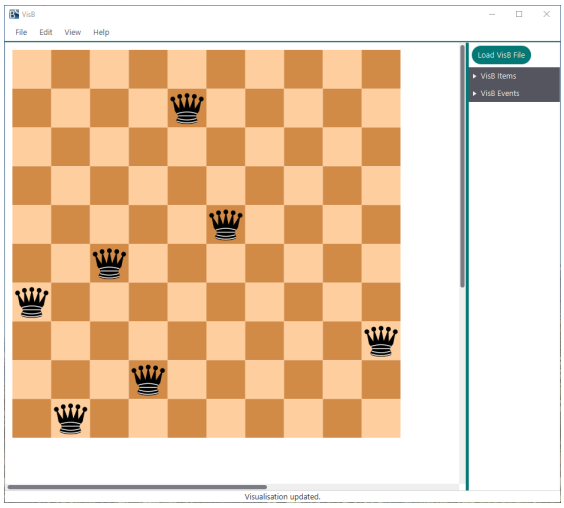


(f) Set the queen in column 1 to row 7.

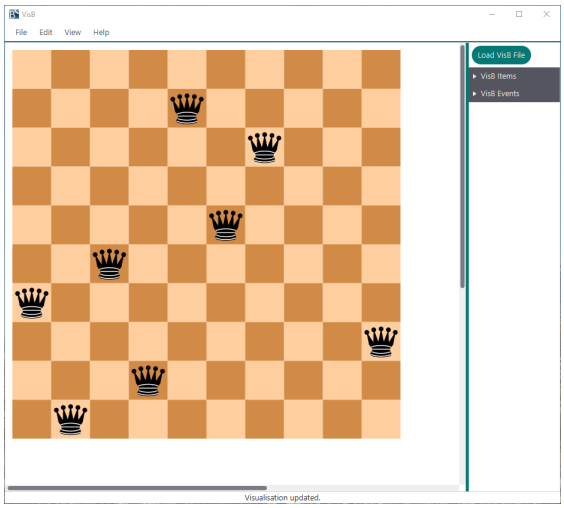
Figure 23: Use Case Example for N-Queens Visualisation Part 1



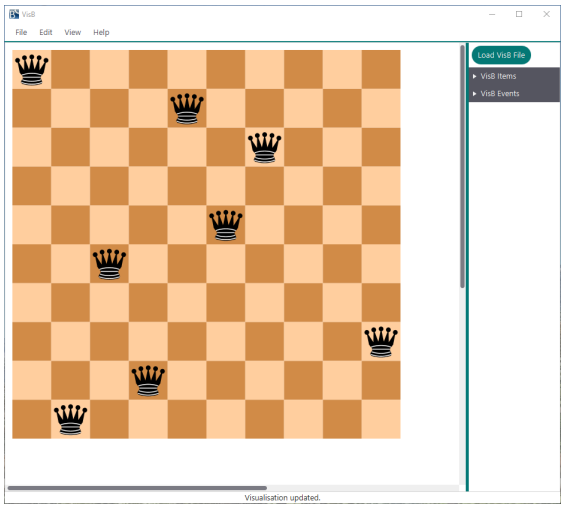
(a) Set the queen in column 5 to row 2.



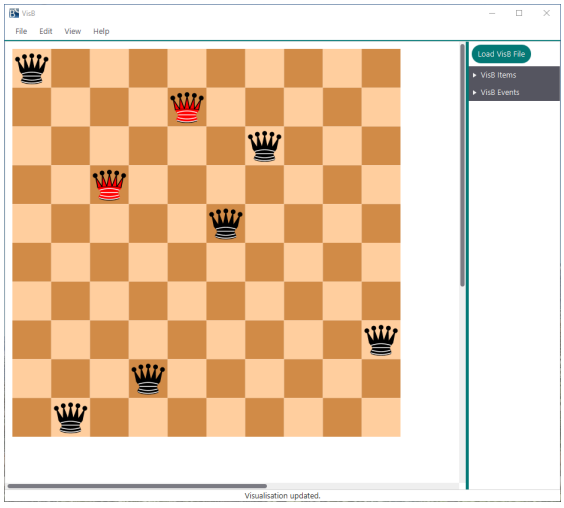
(b) Set the queen in column 10 to row 8.



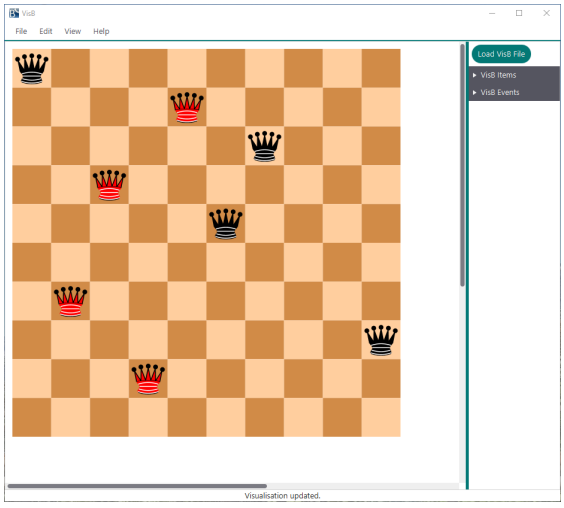
(c) Set the queen in column 7 to row 3.



(d) Change the queen of column 1 to row 1.

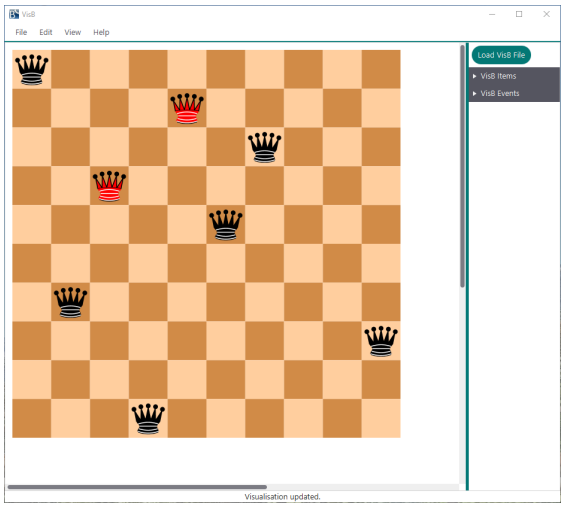


(e) Change the queen of column 3 to row 4.

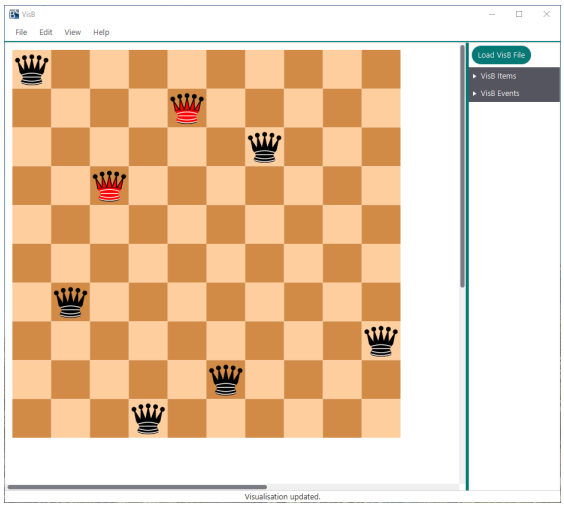


(f) Change the queen of column 2 to row 7.

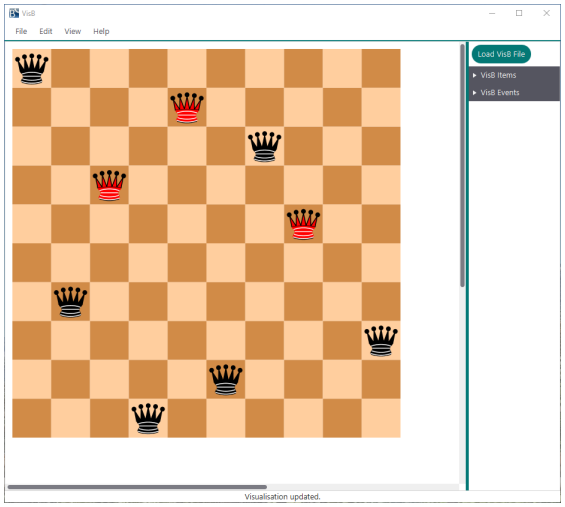
Figure 25: Use Case Example for N-Queens Visualisation Part 2



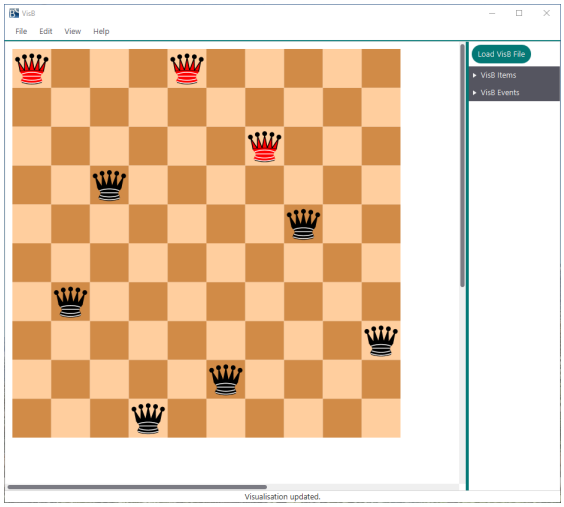
(a) Change the queen of column 4 to row 10.



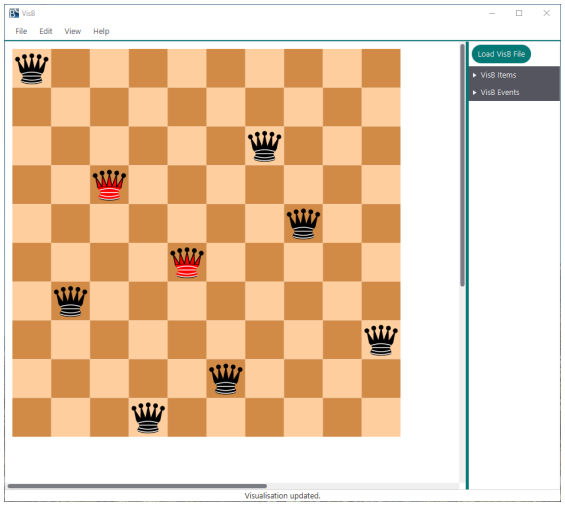
(b) Change the queen of column 6 to row 9.



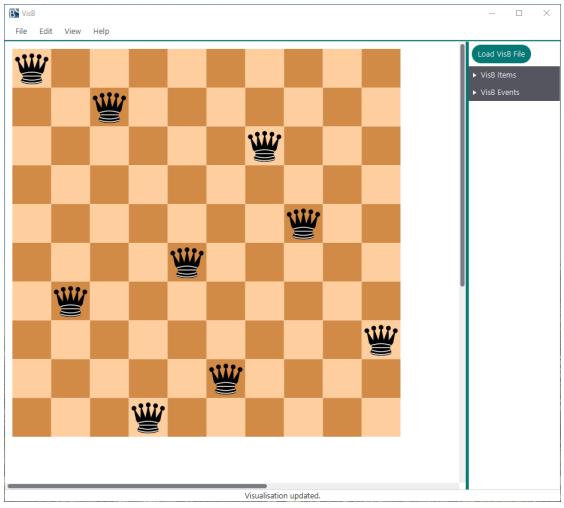
(c) Set the queen in column 8 to row 5.



(d) Change the queen of column 5 to row 1.

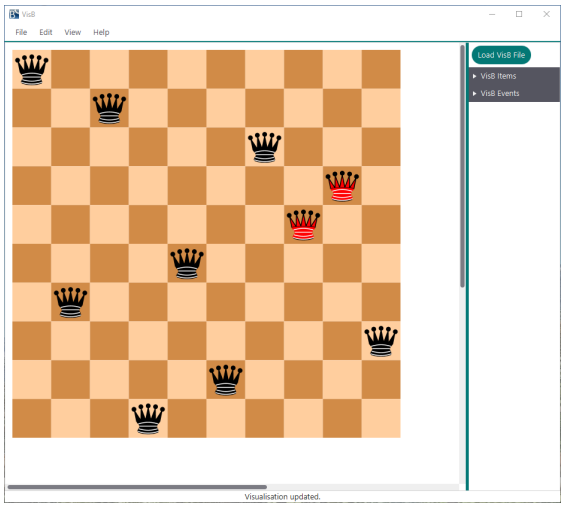


(e) Change the queen of column 5 to row 6.

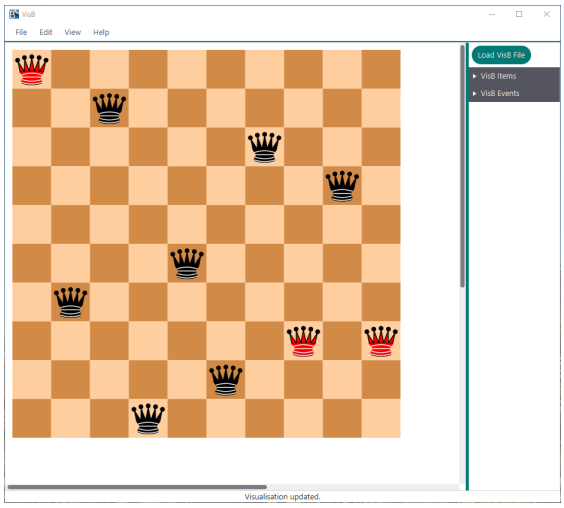


(f) Change the queen of column 3 to row 2.

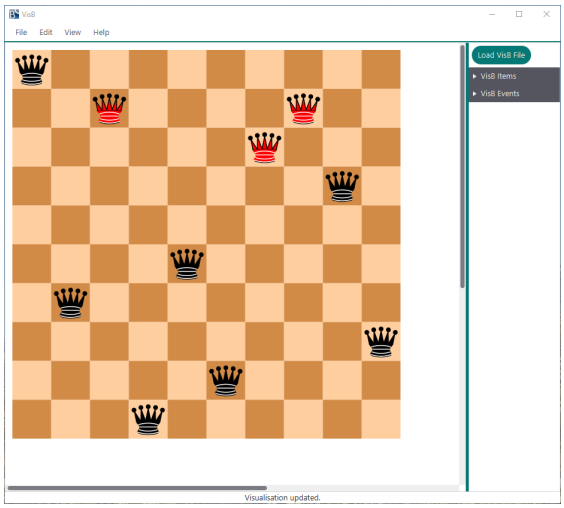
Figure 27: Use Case Example for N-Queens Visualisation Part 3



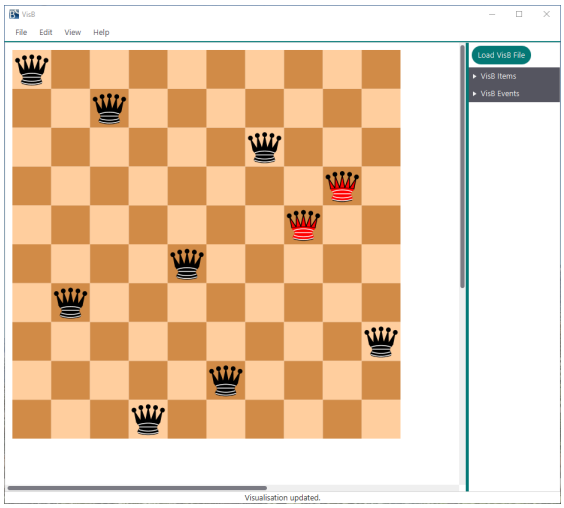
(a) Set the queen in column 9 to row 4.



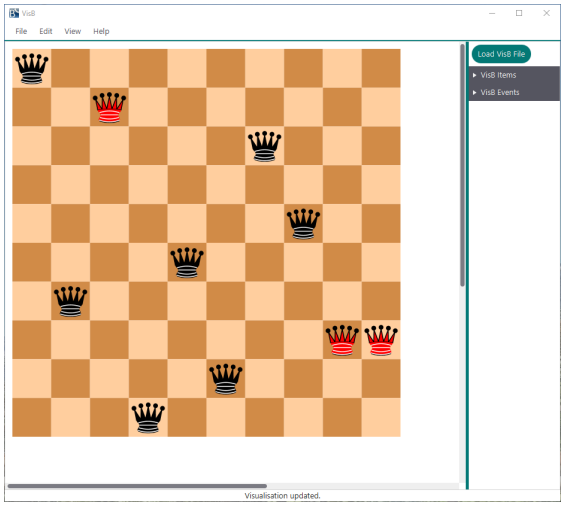
(b) Change the queen of column 8 to row 8.



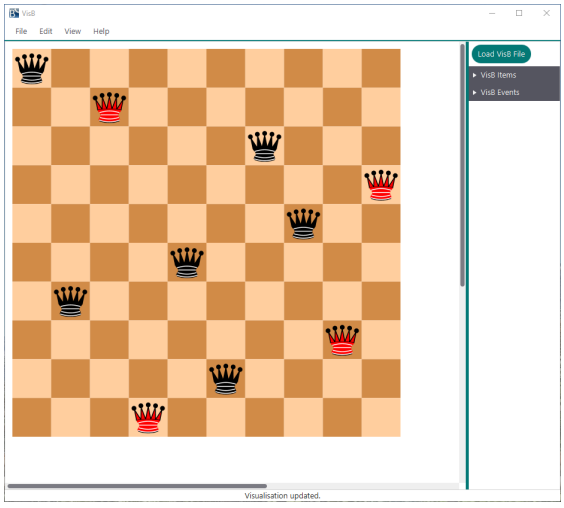
(c) Change the queen of column 8 to row 2.



(d) Change the queen of column 8 to row 5.

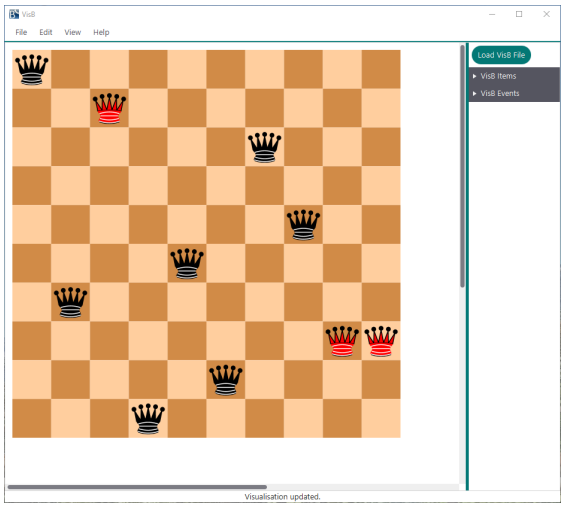


(e) Change the queen of column 9 to row 8.

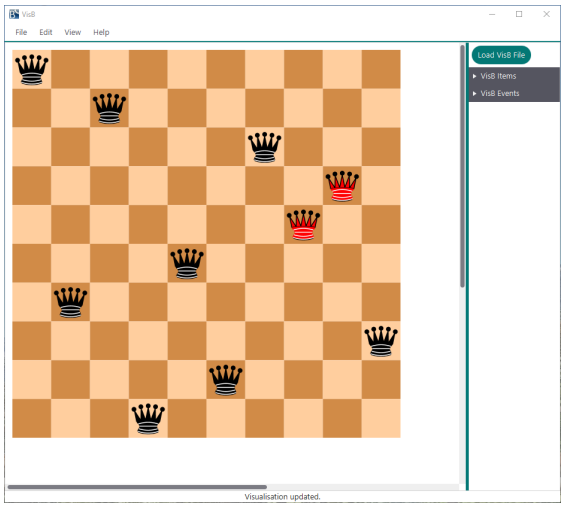


(f) Change the queen of column 10 to row 4.

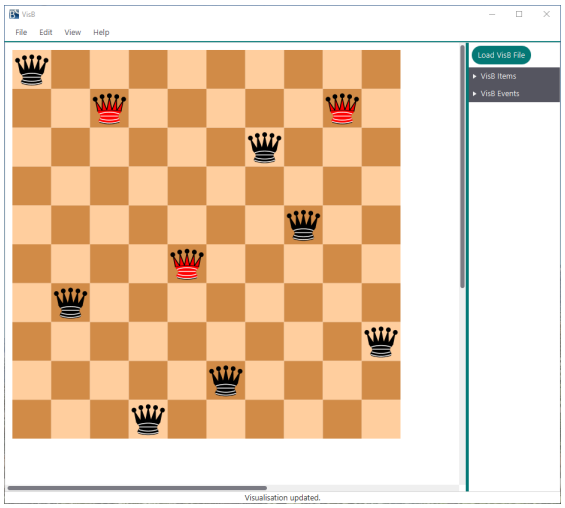
Figure 29: Use Case Example for N-Queens Visualisation Part 4



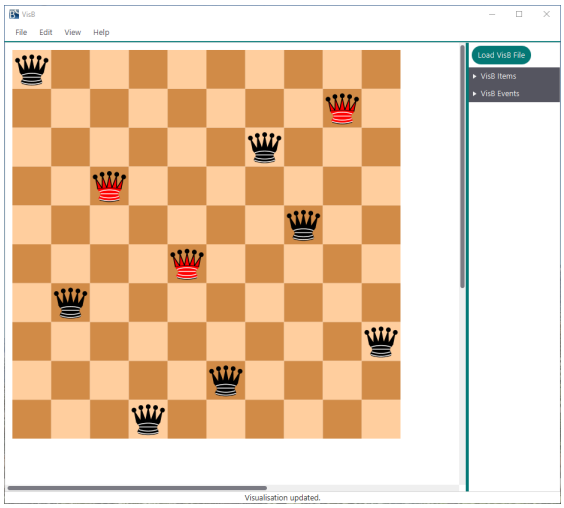
(a) Change the queen of column 10 to row 8.



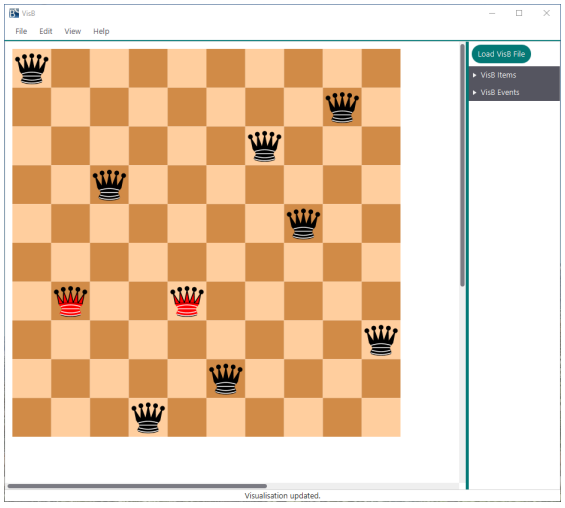
(b) Change the queen of column 9 to row 4.



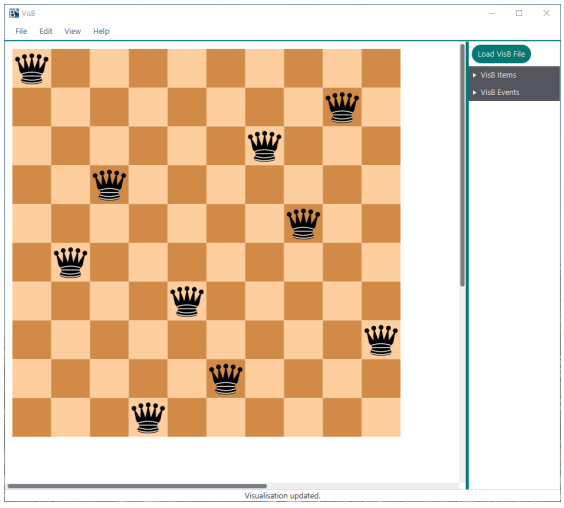
(c) Change the queen of column 9 to row 2.



(d) Change the queen of column 3 to row 4.



(e) Change the queen of column 5 to row 7.



(f) Change the queen of column 2 to row 6.

Figure 31: Use Case Example for N-Queens Visualisation Part 5

C.3 Additional B Model Code Examples

```
1  MACHINE QueensWithEvents
2  // a version of the N-queens model with events to pre-fill the
       chessboard by hand
3  CONSTANTS n
4  PROPERTIES
5   n : NATURAL &
6   n < 120
7  DEFINITIONS
8        ANIMATION_FUNCTION_DEFAULT == ( {r,c,i|r:1..n & c:1..n & i=(r+c)
           mod 2 }  );
9        ANIMATION_FUNCTION == ( {r,c,i|c:1..n & c|->r:queens & i=2+((r+c)
           mod 2) }  );
10       ANIMATION_FUNCTION1 == ( {r,c,i|c:1..n & r:1..n & c|->r/:queens &
           pos_is_attacked(c,r) & i=13+((r+c) mod 2) }  );
11       ANIMATION_FUNCTION2 == ( {r,c,i|c:1..n & c|->r:queens &
           is_attacked(c) & i=10 }  );
12       ANIMATION_IMG0 == "Queens/ChessPieces/Chess_emptyl45.gif";
13       ANIMATION_IMG1 == "Queens/ChessPieces/Chess_emptyd45.gif";
14       ANIMATION_IMG2 == "Queens/ChessPieces/Chess_qll45.gif";
15       ANIMATION_IMG3 == "Queens/ChessPieces/Chess_qld45.gif";
16       ANIMATION_IMG10 == "Queens/ChessPieces/Chess_qrt45.gif"; // Red
           queen
17       ANIMATION_IMG11 == "Queens/ChessPieces/Chess_emptyg45.gif"; //
           green square
18       ANIMATION_IMG12 == "Queens/ChessPieces/Chess_emptyXg45.gif"; //
           green square with X
19       ANIMATION_IMG13 == "Queens/ChessPieces/Chess_emptyDOTl45.gif"; //
           white square with dot
20       ANIMATION_IMG14 == "Queens/ChessPieces/Chess_emptyDOTd45.gif"; //
           black square with dot
21       SET_PREF_TIME_OUT == 6000;
22       SET_PREF_CLPFD == TRUE;
23       SET_PREF_MAX_INITIALISATIONS == 120;
24       SET_PREF_MAX_OPERATIONS == 10;
25       MAX_OPERATIONS_SetQueen == 1000;
26       MAX_OPERATIONS_ChangeQueen == 1000;
27       MAX_OPERATIONS_SolveAny == 4;
28       //SET_PREF_RANDOMISE_ENUMERATION_ORDER == TRUE;
29       ANIMATION_RIGHT_CLICK(I,J) == CHOICE SetQueen(I,J) OR ChangeQueen(
           I,J) OR Solve OR SolveFuzzy END;
30       ANIMATION_CLICK(I,J,tocol,torow) == CHOICE SetQueen(I,J) OR
           ChangeQueen(I,J) END;
```

```
31
32      is_attacked(q1) == q1:dom(queens) &
33                         #q2.(q2:dom(queens) & q2 /= q1 &
34                              (no_attack(q1,q2,queens) => queens(q1) =
                                    queens(q2)));
35      pos_is_attacked(q1,q1row) ==
36                         #q2.(q2:dom(queens) & q2 /= q1 &
37                              (no_attack_pos(q1,q1row,q2,queens(q2)) =>
                                    q1row = queens(q2)));
38      no_attack(q1,q2,board) == no_attack_pos(q1,board(q1),q2,board(q2))
           ;
39      no_attack_pos(q1,q1row,q2,q2row) == (q1row+q2-q1 /= q2row & q1row-
           q2+q1 /= q2row);
40      Solution(board) == (
41         board :  perm(1..n) /* for each column the row in which the
               queen is in */
42         &
43         !(q1,q2).(q1:1..n & q2:2..n & q2>q1 => no_attack(q1,q2,board) )
44        )
45  VARIABLES queens
46  INVARIANT
47    queens : (1..n) +-> (1..n)
48  INITIALISATION
49    queens := {}
50  OPERATIONS
51    Solve = ANY solution WHERE
52        Solution(solution) &
53        !x.(x:dom(queens) => solution(x)=queens(x))
54    THEN
55       queens := solution
56    END;
57    SolveFuzzy = ANY solution WHERE
58        Solution(solution) &
59        !x.(x:dom(queens) => solution(x):{queens(x)-1,queens(x),queens(x)
             +1})
60    THEN
61       queens := solution
62    END;
63    SetQueen(i,j) = SELECT i:1..n & j:1..n & i /: dom(queens) THEN
64       queens(i) := j
65    END;
66    ChangeQueen(i,j) = SELECT i:1..n & j:1..n & i : dom(queens) & j /=
         queens(i) THEN
67       queens(i) := j
```

```
68   END;
69   r<--Get(yy) = PRE yy:dom(queens) THEN r:= queens(yy) END
70 END
```

Listing 19: Animation Function Example from the ProB Public Examples [4]

References

[1] Jean-Raymond Abrial. “Formal Methods: Theory Becoming Practice”. In: *Journal of Universal Computer Science* 13.5 (May 28, 2007), pp. 619–628. URL: http://www.jucs.org/jucs_13_5/formal_methods_theory_becoming.

[2] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[3] Jean-Raymond Abrial and Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[4] Michael Leuschel et al. *ProB Public Examples*. Available at https://www3.hhu.de/stups/downloads/prob/source/ (2019/06/02).

[5] Jens Bendisposto and Michael Leuschel. “A Generic Flash-based Animation Engine for ProB”. In: *Proceedings of the 7th International B Conference (B2007)*. LNCS 4355. Besancon, France: Springer-Verlag, 2007, pp. 266–269.

[6] Pierre Bourhis et al. “JSON: data model, query languages and schema specification”. In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM. 2017, pp. 123–135.

[7] Michael Butler and Michael Leuschel. “ProB: A model checker for B”. In: *International Symposium of Formal Methods Europe*. Springer. 2003, pp. 855–874.

[8] Mathieu Comptier et al. “Property-Based Modelling and Validation of a CBTC Zone Controller in Event-B”. In: *International Conference on Reliability, Safety, and Security of Railway Systems*. Springer. 2019, pp. 202–212.

[9] *Eight Queens Puzzle*. Available at https://en.wikipedia.org/wiki/Eight_queens_puzzle (2019/05/06).

[10] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. *Scalable vector graphics (SVG) 1.0 specification*. iuniverse, 2000.

[11] David Flanagan. *JavaScript: the definitive guide*. " O’Reilly Media, Inc.", 2006.

[12] Dominik Hansen et al. “Using a Formal B Model at Runtime in a Demonstration of the ETCS Hybrid Level 3 Concept with Real Trains”. In: *Proceedings ABZ 2018*. Ed. by Michael Butler et al. Vol. 10817. LNCS. Springer, 2018, pp. 292–306.

[13] Christoph Heinzen. “A user-interface Plugin for the Rule Validation Language in ProB”. MA thesis. 2018.

[14] Christoph Heinzen. *Visualisation Examples*. Available at https://github.com/hhu-stups/prob2-ui-visualizations/ (2019/05/06).

[15] *Introducing JSON*. Available at http://www.json.org/ (2019/18/06).

[16] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[17] Philipp Körner et al. “Embedding High-Level Formal Specifications into Applications”. In: *Proceedings of the 23rd International Symposium on Formal Methods (FM 2019)*. Vol. (to appear). LNCS. Springer, 2019.

[18] Lukas Ladenberger. “Rapid Creation of Interactive Formal Prototypes for Validating Safety-Critical Systems”. PhD thesis. 2016.

[19] Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. “Visualising Event-B models with B-Motion Studio”. In: *Proceedings FMICS’2009*. LNCS 5825. Verlag, 2009, pp. 202–204.

[20] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[21] Michael Leuschel et al. “Easy Graphical Animation and Formula Viewing for Teaching B”. In: *The B Method: from Research to Teaching* (2008). Ed. by C. Attiogbé and H. Habrias, pp. 17–32.

[22] Michael Leuschel et al. *ProB Handbook*. Available at `https://www3.hhu.de/stups/handbook/prob2/prob_handbook.html` (2019/05/06).

[23] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[24] *SVG Path*. Available at `https://docs.oracle.com/javafx/2/api/javafx/scene/shape/SVGPath.html` (2019/05/06).

List of Figures

List of Tables

List of Listings