INSTITUT FÜR INFORMATIK
Datenbanken und Informationssysteme

Universitätsstr. 1 D–40225 Düsseldorf

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Data Visualization in ProB

Joy Clark

Bachelorarbeit

Beginn der Arbeit:	14. März 2013
Abgabe der Arbeit:	14. Juni 2013
Gutachter:	Prof. Dr. Michael Leuschel
	Prof. Dr. Frank Gurski

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 14. Juni 2013

Joy Clark

Abstract

Hier kommt eine ca. einseitige Zusammenfassung der Arbeit rein.

Contents

Contents **i**

1 Introduction **1**

2 Background **1**

2.1 B-Method 1

2.2 ProB 1

2.3 D3 and JavaScript 2

2.3.1 Core Functionality 2

2.3.2 Further Functionality 3

3 Motivation **3**

4 Contribution **4**

4.1 Visualization framework 4

4.2 Visualization of the State Space 5

4.3 Visualization of the Value of a Formula Over Time 6

4.4 Visualization of a formula 7

5 Related Work **7**

6 Conclusion **8**

7 Future Work **8**

References **9**

List of Figures **10**

List of Tables **10**

1 Introduction

During the course of this paper, the different tools and concepts that were necessary in the scope of this work will be introduced. Then the motivation and the requirements for the desired visualizations will be described in detail. The actual visualizations that were created will then be presented, followed by further ideas for implementations and related work.

2 Background

2.1 B-Method

The B-Method is a method of specifying and designing software systems that was originally created by J.R. Abrial [AAH05]. Central to the B-Method are the concepts of *abstract machines* that specify how a system should function [Sch01].

An *abstract machine* describes how a particular component should work. In order to do this, the machine specifies *operations* which describe how the machine should work. An *operation* can change the *state* of the machine. A *state* is a set of *variables* that are constrained by an *invariant*. In order for the model to be valid, the *invariant* must evaluate to true for every state in the model.

Two specification languages used within the scope of the B-Method are Classical B and Event-B. There are several tools available to check and verify these specification languages [LIST TOOLS].

2.2 ProB

ProB is a tool created to verify formal specifications [LB03]. In addition to verifying Classical B and Event-B specifications, ProB also verifies models written in the CSP-M, TLA+, and Z specification languages. ProB differs from other tools dealing with model verification in that it is fully automated. Several tools have also been written to extend ProB and add functionality.

ProB verifies models through consistency checking and animation. Conistency checking is the systematic check of all states within a particular specification. In order to do this, ProB checks the state space of the specification in question. The state space is a graph with *states* saved as vertices and *operations* saved as edges.

ProB is a model checker and animator for specifications written in the Classical B, Event-B, CSP-M, TLA+, and Z specification languages [LB03]. There is a standalone version of the ProB software available with the graphical user interface written in Tcl/Tk. A binary command-line interface is also available for the software.

In 2006, a project began to develop a ProB plug-in for the Rodin software suite so that ProB could be used in conjunction with Rodin.

In the fall of 2011, planning for the ProB 2.0 API began. The main goal of the ProB 2.0

API was to adapt and optimize the existing Java API to build a user interface on top of a programmatic API. Functional programming techniques were used in the development of the software as much as possible. To meet these ends, the Groovy scripting language was heavily integrated into to the ProB 2.0 core. The ProB 2.0 API includes a fully functional webserver with servlets that allow the extension of the Java core into JavaScript and HTML. The fully functional webconsole available in the API makes use of this technology.

2.3 D3 and JavaScript

Since a web server was already available in the ProB 2.0 application, it was plausible to create visualizations using javascript and HTML. The ProB 2.0 application is an Eclipse application, so it also would have been possible to create visualizations using a native Java or Eclipse library. I carried out an experiment at the beginning of this work to determine the feasibility of the different graph libraries. JUNG was considered because it is the software framework that is the ProB 2.0 API currently uses, but it was discarded because of the difficulty of embedding Swing visualizations into Eclipse applications. The ZEST graph library was a feasible option, but in the end, I chose to use the D3 library.

D3 (Data-Driven Documents) is "an embedded domain-specific language for transforming the document object model based on data" which is written in JavaScript [BOH11]. Developers can embed the library into a JavaScript application and use the D3 functions to create a pure SVG and HTML document object model. The focus of D3 is not on creating data visualizations. It is on providing the user the capability of defining exactly which elements the DOM should contain based on the data that the user has provided. Because the objects that are being manipulated are pure SVG and HTML, the user can use D3 to create objects that can be styled using CSS or by dynamically manipulating the style tags of the elements.

2.3.1 Core Functionality

D3 provides a selector API based on CSS3 that is similar to jQuery [jQu11]. The user creates visualizations by selecting sections of the document and binding them to user provided data in the form of an array of arbitrary values [BOH11]. D3 provides support for parsing JSON, XML, HTML, CSV, and TSV files. Once the data is bound to the desired section of the document, D3 can append an HTML or SVG element onto the section for each element of data. This is where the real power of D3 lies because the user can define the attributes of the element dynamically based on the values of the datum in question. By changing these attributes (e.g. size, radius, color) the resulting document already presents the data in a way that the viewer visually understands. The core also provides support for working with arrays and for defining transitions that can be used to animate the document.

2.3.2 Further Functionality

D3 also provides further functionality for manipulating the DOM. Developers can define a scale based on the domain and range of values that are defined in the data provided by the user. The placement of elements within the document can then be placed according to the desired scale. D3 provides support for many different types of scales including linear scales, power scales, logarithmic scales, and temporal scales. Axes can also be created to correspond to the defined scale.

The user has the ability to change the DOM as needed. However, D3 also supports a large number of visualization layouts so that the user does not have to define the positions for the elements in a given visualization. The two layouts that are of relevance for this work are the tree layout and the spring layout.

The tree layout uses the Rheingold-Tilford algorithm for drawing tidy trees [RJT81]. The force layout uses an algorithm created by Dwyer [Dwy09] to create a scalable and constrained graph layout. The physical simulations are based on the work by Jakobsen [Jak03]. The implementation "uses a quadtree to accelerate charge interaction using the Barnes–Hut approximation. In addition to the repulsive charge force, a pseudo-gravity force keeps nodes centered in the visible area and avoids expulsion of disconnected subgraphs, while links are fixed-distance geometric constraints. Additional custom forces and constraints may be applied on the "tick" event, simply by updating the x and y attributes of nodes" [Bos12a].

To help the viewer interact with the visualization, D3 provides support for the zoom and drag behaviors. This listens to the mouse clicks commonly associated with zooming (i.e. scrolling, double clicking) and enlarges the image as would be expected. With this same mechanism, the developer can enable the user to grab hold of the canvas and pan through the image to inspect it closer.

Despite the considerable functions that D3 offers, it is very easy for the user to begin developing with D3. The API is described in detail on the D3 Wiki [Bos12a], and the D3 website [Bos12b] includes an extensive array of examples that new developers can use as a jumping off point. The D3 developer community is very large, so it is easy to find answers to almost every question online.

3 Motivation

The ProB standalone application written in Tcl/Tk uses Dotty as the library to create visualizations of certain data structures within the ProB application. Unfortunately, these data visualizations are completely missing in both the Java APIs. The intention of this work is to inspect the data visualizations that are available in the Tcl/Tk application and recreate these in the Java 2.0 API. The visualizations should not be hard coded, but should use D3 and the existing webserver structure to create a framework so that similar visualizations can be created in the future using the same principles.

Central to the ProB application is the concept of the state space. The state space is a directed multigraph. The states are saved as vertices in the graph and the operations within the graph are saved as directed edges that transition from one state to another. The main purpose of the ProB software is to verify this state space for inconsistencies. For instance, it is possible to use ProB to find states within the graph that violate the invariant for specification. It is also possible to find states from which there are no further operations possible. This is called a deadlock.

The ProB 2.0 API extracts the information about the existing state space from the ProB CLI and saves it in a programmatic abstraction of the state space. This abstraction saves the information about the different states in a graph data structure using the Java JUNG graph library. The state space object already supports the use of Dijkstra's algorithm to find the shortest trace from the root state to a user defined state. This can be used to find traces that show how an invariant violation or deadlock can be found. What is missing, however, is a visualization of the actual state space itself.

Because the state space is a directed multigraph, this visualization problem is not trivial. It was necessary to find a graph library that would be able to draw a complicated graph. Because the state space varies drastically depending on the machine that is being animated, it was also necessary that the graph library be able to handle graphs of all different shapes and sizes.

A useful feature for the visualization of a state space would also be the ability of the user to manipulate the graph. For instance, the Tcl/Tk version of ProB supports the capability for the user to specify a formula and to merge all states for which the formula evaluates to the same result. Similar functionality was desired for the visualization in the ProB 2.0 API. A useful visualization would also allow the user to specify how the graph should be colored.

Although the visualization of the state space was the focus of this work, there were other sets of data for which a visualization would be useful. The ProB Tcl/Tk version supports a useful visualization of B formulas. The user specified formula is broken down into subformulas and colored so as to specify the value of the formula (e.g. if a given predicate evaluates to true at the specified state, the predicate would be colored green). A similar visualization exists in the ProB 1.3.6 API but not in the ProB 2.0 API.

Another useful visualization that falls into the scope of this work was a visualization of the value of a user defined formula over time. No such visualization exists in any of the ProB applications yet, but it was thought that such a visualization would be relatively simple to generate and useful.

4 Contribution

4.1 Visualization framework

One of the main issues that had to be dealt with at the beginning of the development process was the issue of how to integrate the visualizations into the ProB 2.0 API. At the time, the software already contained a functioning web server using Java servlets. Since the visualizations are written using Javascript and the d3 Javascript library, they needed to use the same framework. Because the visualizations needed to react to changes that take place during the animation of a model, they needed to be able to communicate with the ProB kernel. In order to accomplish this, a javascript function is invoked when the HTML page is loaded. This javascript sets up an interval so that the servlet that is responsible for the visualization is polled every 300 milliseconds to see if there are any changes. Both the servlet and the javascript function keep track of a counter that functions as a time stamp. This number is sent back and forth. If the javascript function identifies a discrepency between the numbers, it polls the servlet and then updates the visualization.

A problem quickly arose because the servlet is a singleton object. There is only one servlet responsible for all of the visualizations of a particular type. The solution for this was to generate a unique session id for every visualization. Using this id, HTML content for the visualization is generated. When the HTML content is loaded, it calls the correct javascript function which sets up the polling interval and generates the visualization for the calculated data.

Once I implemented a way to integrate the visualization servlets into the ProB 2.0 application, it was still necessary to implement an easy way for the user to interact with the visualizations. One of the main advantages of the D3 visualization framework is the flexibility for the user. Using the D3 selectors, it is possible for the user to select and change the attributes of any of the elements of the visualization. In order to offer this functionality to the users from within the ProB 2.0 application, we decided that we needed to lift the functionality from the javascript level into the existing groovy console in the Java 2.0 API. This was accomplished by creating a `Selection` object that represents the action that the user wants to carry out in the visualization. Then the `Selection` object is added to the particular visualization and is applied the next time the visualization is redrawn. The `Selection` object was written so that its functionality is similar to what the user would actually write using the D3 library.

```
\\In the Groovy Console:
\\Define an object to tell the visualization to select the
\\ circles with ids #croot and #c1 and color them pink
x = viz.selectAll("#croot,#c1").attr("fill","#f36")

\\Add this selection to the visualization with session id "0"
viz.addToSession("0",x)
```

4.2 Visualization of the State Space

During the preliminary experiments for State Space visualization, several different graph libraries were tested out. D3 was chosen because it could process graphs of relatively large size in a way that was eye pleasing for users. Visualization of the state space uses the Spring layout that is available from the D3 library. Unfortunately, when visualizing state spaces with a very large amount of states and inputting them all at the default intitial position, it took rather long for a good visualization to emerge. Therefore, the FRLayout from the JUNG graph library was used as a static rendering engine to calculate the ideal initial positions for the states to be visualized in the graph.

One of the main problems with the web framework that was discovered at the start of the development process was the problem of how different state spaces should be visualized at the same time. The ProB 2.0 API supports the animation of multiple state spaces at any given time. When a state space visualization is created, it is created using the state space that is currently being animated. When the animation is switched, a new state space visualization can be created using the new state space that is being animated. The problem is that a state space visualization is not static. Since the state space that is being visualized changes over time when states are added into the graph, the visualization also needs to adapt and grow correspondingly. The solution to this is to have the instance of the state space visualization poll the state space regularly to get any new states that have been discovered. The problem occured because the servlet responsible for dealing with the state space was static. When the polling occured, the servlet did not know which instance of the state space was supposed to be polled.

(NOT YET IMPLEMENTED) (ONCE IMPLEMENTED DESCRIBE THE DETAILS OF HOW IT IS IMPLEMENTED) The visualization of the state space is interactive. The user can grab the nodes within the state space and move them around so that they appear exactly as the user desires. Because it the whole visualization is completely written in d3 and Javascript, it is also possible for the user to dynamically change the DOM of the model using the method described in the above section. It is difficult to create a useful visualization of the whole state space because the user not only wants to inspect how the state space appears as a whole but also the individual states within the operation. Using the zoom functionality available in D3, it was possible to take care of both these needs. The main problem was that if the visualization of the nodes was large enough for the user to read the values of the variable at the given state, it would no longer be easy to see the state space as a whole. This was solved by making the text of the node and edge objects very small. If the user wants to inspect a particular node, they can do so by zooming into the visualization. The text is then larger, and the user only needs to see the nodes that are in the direct neighborhood of the node in question. The user can also click on the background of the visualization in order to p(NOT YET IMPLEMENTED) (ONCE IMPLEMENTED DESCRIBE THE DETAILS OF HOW IT IS IMPLEMENTED)an and view the other nodes.

The user can also input Classical B formulas and thereby filter the graph. This uses the algorithms described in [LT05]. The formula is applied to the state space and all states are merged for which the formula evaluates to the same result. The result is a smaller state space that can be viewed by the user.

4.3 Visualization of the Value of a Formula Over Time

During animation in the ProB 2.0 API, the animation steps that have been taken are saved in a trace. A particular formula can take on different values over the course of a trace. In the implementation of the B state space, a particular state is determined by the different values that a variable takes on. Therefore it is particularly interesting to be able to examine the value of a variable over the course of a trace when dealing with a Classical B or Event B formula. Evaluation of a formula for takes place in the ProB prolog core. In the ProB 2.0 API, a feature was implemented that takes the list of all the states that the trace covers and a particular formula and returns the list of the values that the formula takes on in the course of the trace. This feature was used in order to create a line plot of the values that the formula takes on. This works for all formulas that take on either an integer value or a boolean value (IMPLEMENT BOOLEAN VALUE).

4.4 Visualization of a formula

The ProB CLI already supported the functionality of expanding a formula into its subformulas and finding its value at a given state. However, this functionality would only return the subformulas that were directly beneath the desired formula. For the visualization, it was desired that a given formula could be completely expanded and evaluated and then sent to the ProB 2.0 API. This would ensure that performance would not become an issue. It is now possible to register a Classical B formula in the core and then access the expanded and evaluated formula.

In order to implement the visualization, the d3 tree layout was used 1.

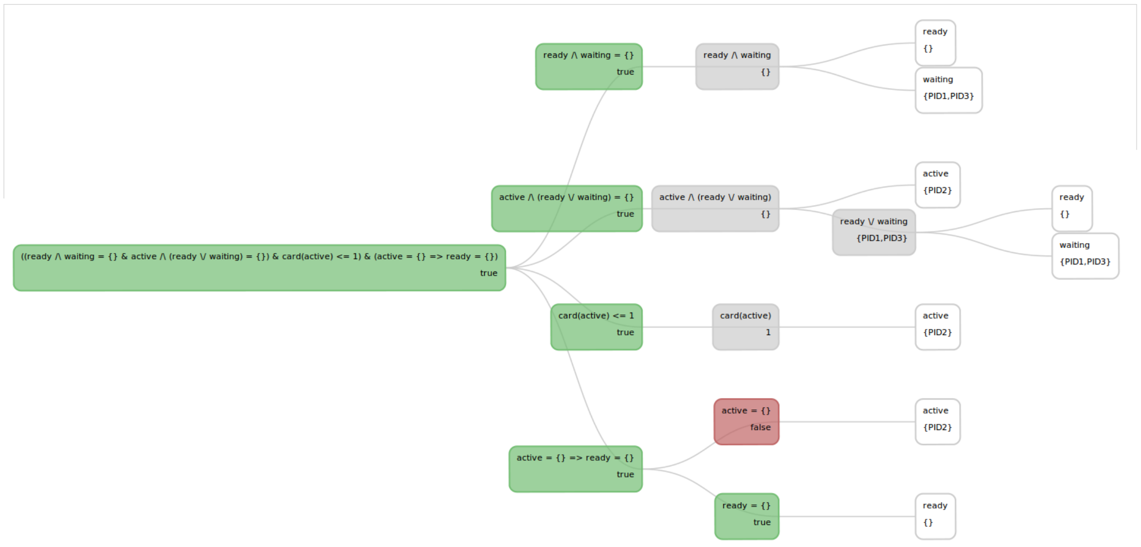

Figure 1: Visualization of the invariant of the Scheduler model

This visualization is interactive. The user can select the nodes to expand or to retract the subformulas. It provides a visualization so that the user can easily interpret the formula. If the formula was evaluated to true for the given formula, the text of the formula is displayed in green. If the formula was evaluated as false, the text is displayed in red. This allows the user to automatically identify the parts of the formula that may have produced the problem.

5 Related Work

d3, Alloy, etc.

6 Conclusion

The Conclusion

7 Future Work

This work has created graphic visualizations using the d3 Javascript library. The same framework can be used to create other visualizations for other needs. For instance, there is no graphical representation of a state in the ProB 2.0 API. One future project could be to create a graphical representation of this state.

It is also possible to adapt the new visualizations in order to add functionality. For instance, in order to view the information about a particular state within the state space, it would make sense to combine the state space visualization with the proposed state visualization. Then, when a state is clicked on within the state space visualization, a window could pop up displaying the state visualization for that particular state.

References

[AAH05] ABRIAL, J.R. ; ABRIAL, J.R. ; HOARE, A.: *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005. – ISBN 9780521021753

[BOH11] BOSTOCK, Michael ; OGIEVETSKY, Vadim ; HEER, Jeffrey: D3: Data-Driven Documents. In: *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2011). `http://vis.stanford.edu/papers/d3`

[Bos12a] BOSTOCK, Michael: *D3 Wiki*. `https://github.com/mbostock/d3/wiki`. Version: 2012

[Bos12b] BOSTOCK, Michael: *Data-Driven Documents*. `http://d3js.org`. Version: 2012

[Dwy09] DWYER, Tim: Scalable, versatile and simple constrained graph layout. In: *Proceedings of the 11th Eurographics / IEEE - VGTC conference on Visualization*. Eurographics Association (EuroVis'09), 991–1006

[Jak03] JAKOBSEN, Thomas: *Advanced Character Physics*. 2003

[jQu11] *jQuery*. `http://jquery.com`. Version: 2011

[LB03] LEUSCHEL, Michael ; BUTLER, Michael: ProB: A Model Checker for B. In: KEIJIRO, Araki (Hrsg.) ; GNESI, Stefania (Hrsg.) ; DINO, Mandrio (Hrsg.): *FME* Bd. 2805, Springer-Verlag, 2003. – ISBN 3–540–40828–2, S. 855–874

[LT05] LEUSCHEL, Michael ; TURNER, Edd: Visualising Larger State Spaces in ProB. In: TREHARNE, Helen (Hrsg.) ; KING, Steve (Hrsg.) ; HENSON, Martin (Hrsg.) ; SCHNEIDER, Steve (Hrsg.): *ZB* Bd. 3455, Springer-Verlag, November 2005. – ISBN 3–540–25559–1, S. 6–23

[RJT81] REINGOLD, Edward M. ; JOHN ; TILFORD, S.: Tidier drawing of trees. In: *IEEE Trans. Software Eng* (1981)

[Sch01] SCHNEIDER, S.: *The B-Method: An Introduction*. Palgrave Macmillan Limited, 2001 (Cornerstones of Computing Series). – ISBN 9780333792841

List of Figures

1 Visualization of the invariant of the Scheduler model 7

List of Tables