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

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 [LB08]. 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 [LB03] and refinement checking [LB05]. 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. Refinement checking examines the refinements of a machine to ensure that they are valid refinements.

2.2.1 ProB Tools

ProB consists of several different tools that will be referenced throughout the course of this work.

ProB CLI

The ProB kernel is written in primarily in SICStus prolog [LB08]. The ProB CLI is available as a binary executable, and all of the other tools listed here are built on top of it. The ProB CLI provides support for the interpretation of Classical B, Event-B, CSP-M, TLA+, and Z specification languages. These specifications are interpreted and translated into an internal represenation that can then be animated and model checked.

ProB Tcl/Tk

The ProB Tcl/Tk was created at the same time as the ProB CLI to provide a user interface for the ProB CLI. In addition to providing a UI for the ProB CLI, ProB Tcl/Tk also enables the user to edit specification files before animation, and provides the user with visualizations of different data that is generated during animation or model checking. For instance, ProB Tcl/Tk includes graphical visualizations for the state space, the current state, and for B predicates that are evaluated at the current state in the animation [LSBL08] The ProB Tcl/Tk application uses the DOTTY tool available from the Graphviz graph layout software.

ProB Plugin

Work on the ProB Plugin began in 2006. This is an Eclipse plugin for the Rodin software [BH07], which is an easy to use and extensible tool platform for editing specifications written in the Event-B specification language. At this point, a socket server was integrated into the ProB CLI which allowed the ProB Plugin, which is written in Java, to communicate with the ProB CLI. The communication between the two takes place using queries and answers.

ProB 2.0 API

Development of the ProB 2.0 API began in 2011. 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. One of the main improvements made available in this tool was the introduction of a programmatic abstraction of the state space. The ProB 2.0 API also provides a programmatic abstraction called a *history* for the represenation of animations. This history consists of the trace of operations that have been executed for a given animation. A given state space can have an arbitrary number of animations. The user can switch between animations and work on any given animation at any given time. Thus, for the ProB 2.0 API, the notion of a *current state* corresponds to the current states of the animation that the user is currently executing.

During the course of animation and model checking, the state of the ProB 2.0 API changes. The state space grows (i.e. new operations and states are cached), and the current state changes. In order for developers to be aware of these changes, a listener framework is offered. A developer can therefore implement classes that react when the current state in the animation changes, when new states are added to the state space, and when the specification that is being animated changes.

The API is programmatic and harnesses the power of the Groovy programming language. The ProB 2.0 API integrates a fully functioning groovy console into the final product. It is now possible for users and developers to write Groovy scripts that carry out desired functionality. There is also support for creating web appli-

cations that communicate with ProB. The console is actually a user interface that makes use of the web server that is integrated in the ProB 2.0 API. There are no eclipse dependencies present in the ProB 2.0 API, so it can be deployed as a jar file and integrated into any Java based application.

ProB 2.0 Plugin
Similar to the original ProB Plugin, the ProB 2.0 Plugin is an Eclipse plugin created for the Rodin platform. Much of the UI code has been directly imported from the original Plugin, so it appears to be very similar. However, the graphical interface is now built on top of the new programmatic abstractions that are available from the ProB 2.0 API, and changes that take place within the graphical interface are triggered by the ProB 2.0 API listener framework.

2.3 D3 and JavaScript

Since a web server was already available in the ProB 2.0 Plugin, it was plausible to create visualizations using javascript and HTML. Because the ProB 2.0 Plugin is an Eclipse application, 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 Tcl/Tk application uses the DOTTY graph layout tool to create visualizations of data structures within the ProB application. Unfortunately, these data visualizations are completely missing in both the Java applications. In this work, we intend 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 all use D3 and the existing webserver structure. After we are done implementing a few chosen visualizations, it should be possible to create similar visualizations using the principles that we have used.

The concept of the state space is central to the ProB application. 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.

3.1 Planned Visualizations

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 because the algorithm for drawing graphs is not trivial. We had to find a graph library that would be able to draw a such a 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. We chose D3 because it met all of these requirements.

We also wanted the user to be able to manipulate and interact with the graph. For instance, the ProB CLI supports the ability to create smaller graphs that are derived from the original state space [LT05]. Because state spaces become so large so quickly, a derived graph can be much more meaningful and useful for a user. One of the features that we want to implement for the state space visualization is the ability for the user to apply these algorithms to their state space to simplify its representation. The calculation of these algorithms needs to take place in the ProB CLI, but a seamless transfer between the two graphs should take place from within the visualization.

Although the visualization of the state space is the focus of this work, there are other sets of data that need to be visualized. The ProB Tcl/Tk version supports a useful visualization of B formulas [LSBL08]. 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 Plugin, but not in ProB 2.0.

We also wanted to create a visualization of the value of a user defined formula over the course of an animation. No such visualization exists in any of the ProB applications yet, but we thought that such a visualization would be an easy one to implement and would provide a way to quickly test out the D3 framework to ensure that it is a good choice for our graphical visualizations.

3.2 Integration into the ProB 2.0 Eclipse Plugin

Because ProB 2.0 is an Eclipse plugin, it is necessary to be able to integrate any visualization into the framework. Because D3 is JavaScript based, we will create Eclipse views that contain a browser displaying the visualization that we create. A similar technique has already been used to create the groovy console view in the ProB 2.0 application.

It is also necessary that the Java servlets responsible for generating and manipulating the visualizations can manage several different visualizations at once. We have not integrated any server side language into our webserver, so the visualizations need to consist only of static html pages and the JavaScript programs that are responsible for generating the visualization. Data will not be able to be sent to any of the visualizations directly. Instead, the JavaScript scripts will have to set up an interval to poll the servlet and ask for any changes in their visualization.

It should also be possible for the user to edit the visualization. Using D3 selectors, it is possible to change the style of D3 visualizations. We should find a way to lift this functionality from the JavaScript level into the Java application so the user has as much control over the generated visualization as possible.

4 Contribution

4.1 Visualization of the State Space

When a state space visualization is opened, the visualization servlet responsible for the state space visualization takes the state space associated with the current animation and extracts the information about the nodes and edges contained within the graph. This information is then processed by D3 using the force layout. Unfortunately, the state space contains an extremely large amount of information that has to be processed. This includes:

1. How the state space graph appears as a whole.
2. The values of the variables for every state in the graph.
3. The names and parameters of the operation the corresponds with every edge in the graph.
4. The value of the invariant for every state in the graph.
5. Whether or not a given node within the graph is deadlocked.
6. Multiple edges between given states.
7. Looped edges to one given state.
8. The direction of any given edge.
9. A special representation to show the root node within the graph.

Because of all of these things, it was rather 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. To solve this problem, we used the zoom functionality that is available in D3. 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 possible to see the state space as a whole. Instead of trying to meet both requirements at once, we simply made the text that is printed on the node and edge objects very small. When the visualization is created, the user can inspect the graph as a whole how the graph appears as a whole. The text for the given nodes, however, is virtually indiscernable. 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 can see the nodes that are in the direct neighborhood of the node in question. Then user can also click on the background of the visualization in order to pan through the visualization and inspect other nodes and edges.

The visualization of the edges was also not trivial. For most implementations of the force layout, developers use lines as edges. The problem with lines is that if multiple edges occur between two nodes (which happens almost always in a state space), the edges within the representation will be drawn on top of each other. To avoid this, it was necessary to use the SVG path object instead of the SVG line object. Then the paths are drawn with a curve and they do not appear on top of each other. It was also necessary to find a way to determine how many edges exist between two nodes, because if the arcs are generated statically, there is still a good possibility that they will be drawn on top of each other. It was also difficult to figure out how to draw self-loops. By default, self-loops are not drawn in force graphs, so it was necessary to find a way to do so.

We also had some performance issues that were associated with very large state spaces. The force layout keeps adjusting the graph until it reaches a fix point. The problem was that as the state space grew, there were more and more objectst that had to be accounted for. The force layout just kept calculating and moving the the nodes. This didn't only affect the appearance of the visualization. It cost enough resources that the whole eclipse plugin would become unresponsive.

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. 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.

4.2 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, the expanding of the formula took place lazily. A formula would be sent to the ProB CLI and then the direct subformulas of this formula would be sent back. The software would then have to contact ProB CLI recursively until all of the subformulas had been calculated and cached on the Java side. For the predicate visualization, we wanted the formula to be completely expanded and then sent in its entirety to the ProB 2.0 API. In order to do this, we implemented a prolog predicate within the ProB CLI that performs the recursive expanding of a B formula before it is sent back to the Java API. The predicate also delivers the value of each subformula for the given state. This ensures that performance will not an issue.

The final visualization is interactive 1. If a formula has subformulas, the user can select it from within the visualization to expand or to retract the subformulas. The subformulas are always either predicates or expressions. If they are expressions, they are colored white or light grey depending if they have subformulas or not. If the formula is a predicate that has evaluated to true for the given formula, the node is colored green. If the formula is a predicated that has evaluated to false, node is colored in red. The value of the given formula is also printed beneath the formula. This allows the user to visually identify the parts of the formulas and their given values.

In the implementation of the formula, the D3 tree layout is used. The expanding and collapsing of the nodes takes place with a simple JavaScript function. By harnessing the power of the D3 zoom behavior, it is also possible to zoom in and out of the visualization and to pan the image to inspect it closer. The servlet responsible for the visualization implements a listener to identify if any changes in the animation occur. If they do, the formula is recalculated for the new current state, and the visualization is redrawn.

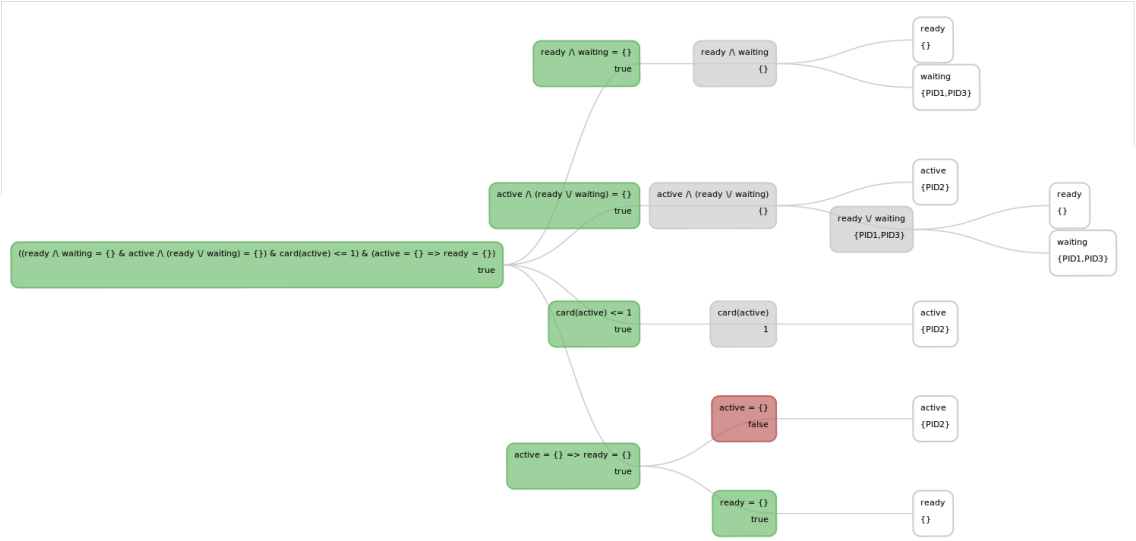

Figure 1: Visualization of the invariant of the Scheduler model

4.3 Visualization of the Value of a Formula Over Time

The ProB 2.0 API supports the evaluation of a given formula over the course of the history of an animation. Because a state is defined by the values that the variables take on when dealing with B type specification languages, it can be 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.

In order for the ProB 2.0 API to evaluate a formula, it extracts the list of states that the animation visits over the course of its history. Then it contacts the ProB CLI and extracts the value that the formlua takes on for each state in the list. This information is then processed by D3 to produce a simple line plot 2. As of now, formulas can only be visualized if they take on integer values. In the future, we plan to support the visualization of formulas that take on boolean values. The visualization also interacts with the ProB 2.0 API. If the current state changes, the formula is recalculated and a new plot is produced.

4.4 Visualization framework

One of the main issues that we had to be deal 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

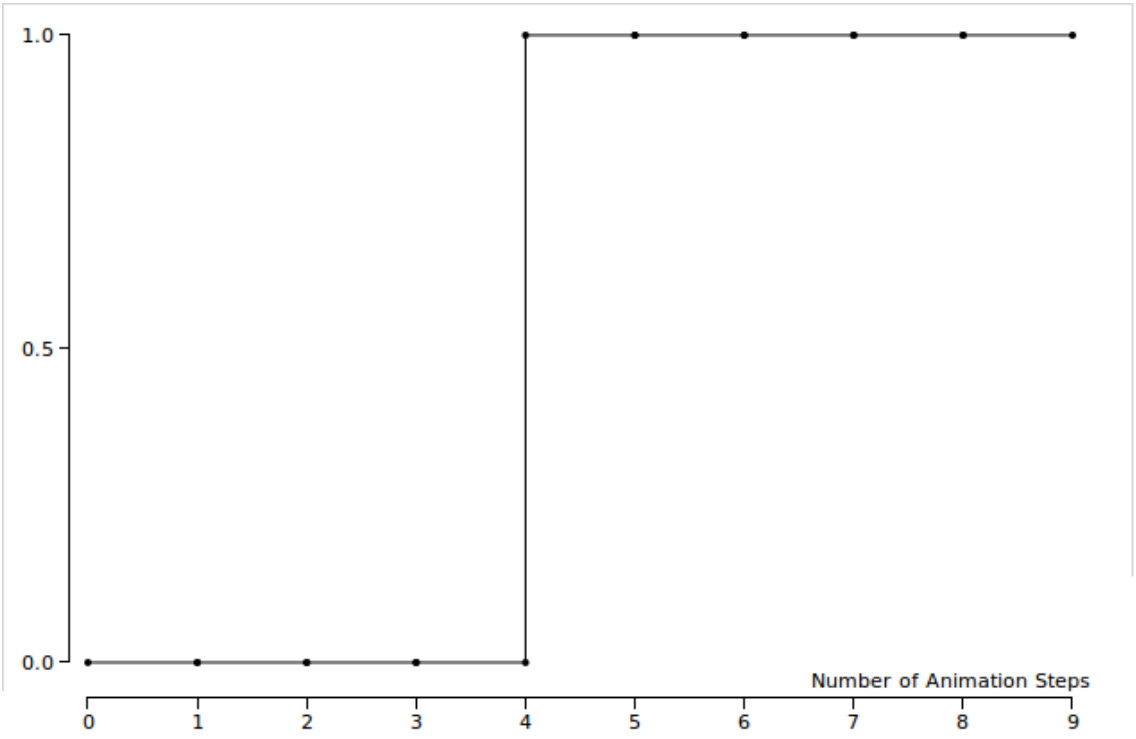

Figure 2: Visualization of value of *card(active)* from the Scheduler model over the course of an animation.

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. However, a static html page will always start with exactly the same values, and there is no good way for the javascript instances to determine what visualization they should belong to based on the content in the html page. The solution for this was to generate a unique session id for every visualization. Using this id, an HTML page for the visualization is generated containing this unique identification number. When the HTML content is loaded, it calls the initialize function in the JavaScript script with the identification number as an id. The JavaScript then sets up a polling interval and generates the visualization for the calculated data.

Once we 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 `Transformer` object that represents the action that

the user wants to carry out in the visualization. Then the `Transformer` object is added to the particular visualization and is applied the next time the visualization is redrawn. The `Transformer` object was written so that its functionality is similar to what the user would actually write using the D3 library.

(NOT YET IMPLEMENTED. BUT THIS IS HOW IT SHOULD BE IMPLEMENTED.) We wanted how the user interacted with the visualization to be as natural as possible. The user should not have a hard time learning how to manipulate the visualization. For this reason, we decided to create a small DSL that would enable the user to specify which attributes should change within a particular visualization. When a visualization session is created, the visualization handler creates a variable within the groovy console. In this variable, an object is saved that communicates with a particular visualization servlet (e.g. the state space servlet) and tells the servlet that it wants to change itself. To do this, we used the functionality offered by Groovy Builders.

```
\\In the Groovy Console:
\\Tell the visualization corresponding to viz1 that it
\\should create a transformer with selection "#rroot,#r1".
\\It should then fill in the rectangles with the color red.
x = "#rroot,#r1" {
    "fill" : "red"
    "stroke" : "gray"
}

viz0.apply(x)
```

The ability to change visualizations in this way is built into all of the visualizations. It is therefore possible to manipulate all of the visualizations by changing the attributes that they contain.

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

[BH07] BUTLER, Michael ; HALLERSTEDE, Stefan: The Rodin Formal Modelling Tool. In: *BCS-FACS Christmas 2007 Meeting*, 2007

[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

[LB05] LEUSCHEL, Michael ; BUTLER, Michael: Automatic Refinement Checking for B. In: LAU, Kung-Kiu (Hrsg.) ; BANACH, Richard (Hrsg.): *Proceedings ICFEM* Bd. 3785, Springer-Verlag, May 2005, S. 345–359

[LB08] LEUSCHEL, Michael ; BUTLER, Michael: ProB: An Automated Analysis Toolset for the B Method. In: *Software Tools for Technology Transfer (STTT)* 10 (2008), Nr. 2, S. 185–203

[LSBL08] LEUSCHEL, Michael ; SAMIA, Mireille ; BENDISPOSTO, Jens ; LUO, Li: Easy Graphical Animation and Formula Viewing for Teaching B. In: ATTIOGBÉ, C. (Hrsg.) ; HABRIAS, H. (Hrsg.): *The B Method: from Research to Teaching*, Lina, 2008, S. 17–32

[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

List of Tables