INSTITUT FÜR INFORMATIK
Softwaretechnik und
Programmiersprachen

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:	Dr. Michael Leuschel
	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

The ProB tool performs the animation and verification of formal models through consistency checking and animation. Here we propose a method for the generation of dynamic visualizations to help users better understand the results that are produced from ProB. In order to do this, we used the D3 JavaScript library to generate visualizations based on the data produced by ProB. In this work, we will present three new visualizations that are available for the user:

1. The visualization of the state space graph for the formal model that is being verified. This visualization also allows the user to apply graph reduction algorithms to the state space to create a derived graph that is easier to read.
2. The visualization of a formula and its subformulas evaluated at the current state of an animation.
3. The visualization of the value that a formula or formulas take on over the course of an animation.

We will also describe the new visualization framework that has been implemented to allow the user to interact with the visualizations and to allow the visualizations to be updated when changes in ProB take place. The framework also allows the user to customize the appearance of the visualization. We have found that the integration of these dynamic visualization have improved the overall user experience of ProB. The user can now interactively manipulate data in order to better understanding of the workings of a formal model.

Contents

1 Introduction

ProB is a tool created for the animation and verification of formal methods, in particular for models that are created using the B-Method [1]. The B-Method is a method of specifying and designing software systems that was originally created by J.R. Abrial [2]. Central to the B-Method are the concepts of *abstract machines* that specify how a system should function [3]. Specifications created using the B-Method are written in Classical B or Event-B notation. One of the main use cases for the B-Method is to model and verify that safety-critical software is correct. ProB also provides support for the verification of models written in the CSP, TLA+, and Z specification languages.

In order to verify a formal specification, ProB simulates the state space that corresponds to that model. A *state space* is a labeled transition system that is represented as a directed multigraph. The vertices in the graph represent every possible state in the system. The edges in the graph represent all of the transitions between any given states. ProB verifies a model primarily by performing *consistency checking*. This is the systematic check of all states that are accessible from the initial state. A model is found to be correct if its state space contains no deadlocks (i.e. all vertices in the graph have at least one outgoing edge) and if no state violates the *invariant* for the model. The *invariant* is a predicate that describes the correct behavior of the model and that must evaluate to true for every state in the model in order for the model to be correct.

The user can also use ProB to perform animation of a particular model. *Animation* here refers to the user's ability to interactively select transitions and thereby create a path throughout the state space. When animating, the concept of a *current state* becomes important. The *current state* refers to the state in the state space that the user has reached after executing all of the transitions that the user has specified during animation. It is often of interest to the user to be able to examine elements of the model at the current state in the animation.

During the course of model verification, ProB produces a great deal of data. The purpose of this work is to dynamically generate visualizations based upon this data. The approach for data visualization here differs from the method that is applied in B-Motion studio [4], which allows users to build visualizations that then are updated when the current state in an animation changes.

1.1 Visualization of the State Space

Since the concept of the state space is so central to ProB, the focus of this work will be on generating a visualization of the state space. Algorithms for drawing graphs are not trivial; hence this visualization problem is not trivial. However, the focus of this work is in applying existing graph algorithms to create visualizations as opposed to writing a new algorithm or modifying an existing algorithm. Therefore, it is necessary to find an existing library that is able to take care of the drawing of the state space.

ProB already includes support for the creation of a graph description of the state space in the DOT graph description language. These description files can then be visualized

with GraphViz[1]. However, the graph algorithms available in GraphViz are relatively inefficient. During verification, the state space often grows exponentially. For state spaces that have a very large number of vertices, the GraphViz algorithms take too long to be useful. The graph generation in GraphViz is also inherently offline. If any thing changes in the state space, the whole graph must be rerendered. Consequently, the generated graphs can be very different from each other, and the user can lose track of the area of the state space that is of interest to him. During consistency checking and animation, states are constantly being added to the graph. We therefore want a graph engine that can easily add vertices to an existing visualization and that can handle state spaces with a large number of vertices.

We also want the user to be able to manipulate and interact with the graph. For instance, ProB supports the ability to create smaller graphs that are derived from the original state space [5]. Because state spaces can become so large so quickly, a derived graph can be much more meaningful and useful for a user. In our visualization of the state space, we have enabled the user to apply these algorithms to their state space to simplify its representation. The user can also seamlessly transfer between the representation of the whole state space and the derived graphs.

1.2 Other Visualizations

Although the visualization of the state space is the focus of this work, there are other visualizations that can be generated which will be useful for the user. For instance, ProB supports the generation of a DOT file which, when rendered, shows how a given formula is broken down into subformulas [6]. The formula and its subformulas are also evaluated for a given state, and the resulting represenation of the formula is colored so as to specify its value (e.g. if a given predicate evaluates to true at the specified state, the predicate is colored green). During the course of this work, we also have recreated this visualization using the new graph engine.

Currently, one of the points of interest in the development of ProB is the simulation of time. This can be the simulation of models with a temporal element or the simulation of multiple models concurrently. When this is the case, the user is less interested in the state space for the given model and more interested in being able to see what value that a model element takes on over the course of a given animation. We have therefore also created a line chart to visualize this information.

1.3 Integration of the Visualizations into ProB

The library that we have chosen to take care of the generation of our visualizations is the JavaScript library D3. D3 provides a modern approach to data visualization. Instead of defining a unique grammar and a rendering engine to draw visualizations, D3 provides the user with the ability to create and position HTML and SVG elements within an HTML document. The resulting images can then be rendered by any modern browser and can

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

be styled using CSS. Because D3 is a JavaScript library, it is also possible to update the visualizations dynamically.

The visualizations that are produced using D3 are integrated into the ProB 2.0 Java application using the internal Jetty server that is present. The visualizations also make use of the existing listener framework that is triggered when changes in the state space or the current animation take place. It is also possible to create and view multiple visualizations at any given time.

2 Related Work

The focus of this work was to apply existing graph algorithms to the problem of visualizing a state space. This is the approach currently taken by the ProB software, although we are proposing changing the graph engine from GraphViz[1] to a JavaScript powered visualization. CSP-M[7] is another tool which generates DOT file to visualize the state space associated with CSP specifications.

Another approach to visualizing state spaces is the application of different algorithms to the state space in order to simplify it without losing key characteristics of the state space. Several of these algorithms have already been integrated into ProB [5]. In the course of this work, we have integrated support for two of the algorithms present in ProB. However, there are other algorithms available in ProB, such as the DFA-Abstraction algorithm presented in [5], which have not been supported in this work.

There are also several tools developed for the specific purpose of visualizing state spaces. Van Ham et al. [8] have researched the problem of visualizing labeled transition systems and developed the LTSView[2] tool for visualizing the structure of state spaces. This tool produces a 3D representation of a state space that the user can inspect. In order to simplify the visualizations that are created by LTSView, work has been done to develop a method of converting 3D models of labeled transition systems to 2D [9]. The StateVis tool is the result of this research[3]. These tools can help users to get a rough idea of the feel of the whole state space. However, they do not allow for a close inspection of interesting parts of the state space. In order to provide a solution to this problem, Pretorius and van Wijk have proposed a method for interacting with the visualization of state transition graphs [10]. The tools NoodleView[4] and its successor DiaGraphica[5] attempt to apply this method.

[2] http://www.mcrl2.org/release/user_manual/tools/ltsview.html
[3] http://www.win.tue.nl/vis1/home/apretori/statevis/
[4] http://www.comp.leeds.ac.uk/scsajp/applications/noodleview/
[5] http://www.comp.leeds.ac.uk/scsajp/applications/diagraphica/

3 Background

3.1 ProB 2.0

ProB is written in primarily in SICStus prolog [11]. This application is packaged as a binary executable command line interface, and several other tools have been built on top of it to provide the user with a graphical user interface. The user interface for the current standalone ProB application is written in Tcl/Tk. The GraphViz rendering engine has also been integrated into this application, and it is for this application that all of the existing data visualizations have been created.

ProB 2.0 is another tool that is built on top of the ProB command line interface. It is the successor of an Eclipse RCP plugin that has been in development since 2005 [12]. This application was created so that ProB could be integrated with the Rodin software, which is a tool platform for editing specifications written in the Event-B specification language.

In 2011, development for ProB 2.0 began. The main goal of ProB 2.0 was to adapt and optimize the existing Java application to produce a programmatic API. One of the main improvements that was made in this tool was the introduction of a programmatic abstraction of the state space. It also provides a programmatic abstraction for the represenation of animations. This abstraction consists of the trace of transitions that have been executed during the course of an animation and allows the user to move forward and backwards within the trace. This also provides the user with the concept of a current state. ProB 2.0 also contains a listener framework that is triggered whenever changes take place within a state space or whenever the current state in the animation changes.

The core of the application includes a fully-functional Groovy execution engine with a corresponding Groovy REPL. It is now possible for users and developers to write Groovy scripts that harness the power of ProB. There is also support for creating internal web applications that communicate with ProB.

The graphical user interface in ProB 2.0 is an Eclipse RCP Plugin that is built on top of the programmatic core and provides the interface with which the application can communicate with the Rodin platform. The visualizations that we have created in the course of this work are integrated into ProB 2.0. These visualizations make use of the available web framework within the ProB 2.0 core. Each visualization is a web application written in JavaScript and HTML that communicates with the internal web server in order to receive the data that needs to be visualized. To integrate these visualizations into the Eclipse application, we have loaded them into a browser widget.

3.2 D3 and JavaScript

Since a jetty server was already available in ProB 2.0, it was plausible to create visualizations using JavaScript and HTML. Because ProB 2.0 is an Eclipse application, it also would have been possible to create visualizations using a native Java or Eclipse library. We carried out an experiment at the beginning of this work to determine the feasibility of the different graph libraries. At the time, we were using JGraphT as the graph library for handling the interaction with the state space abstraction. JGraphX, the library responsi-

ble for visualizing the graphs modelled in JGraphT, proved to be incapable of visualizing a directed graph in a way that was pleasing to the eye. We then switched graph libraries to JUNG. The visualizations that JUNG produces are quite nice, and it would have been relatively simple to embed the visualizations into the existing ProB 2.0 application, but customizing JUNG graphs is quite difficult and it would not have been possible to update the graph visualization dynamically. The ZEST graph library was also considered, but in the end we chose to use the D3 library because it is easier to dynamically update visualizations that are powered by D3.

D3 (Data-Driven Documents) is a JavaScript library that is "an embedded domain-specific language for transforming the document object model based on data" [13]. Developers can embed the library into a JavaScript application and use the D3 functions to create a pure SVG and HTML document object model (DOM). 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 attributes of the elements.

3.2.1 Core Functionality

D3 provides a selector API based on CSS3 that is similar to jQuery[6]. 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 [13]. 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, position) 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. In order to better understand how D3 works, we have provided a simple example of a how a developer can use D3 to create an HTML dropdown menu (see Listing 1). The generated HTML snippet is also provided (see Listing 2). A more complicated example from the D3 website[7] that uses the force layout is available in the appendix (see Appendix A).

3.2.2 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

[6] http://jquery.com
[7] http://d3js.org

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 force layout.

The tree layout uses the Rheingold-Tilford algorithm for drawing tidy trees [14]. The force layout uses an algorithm created by Dwyer [15] to create a scalable and constrained graph layout. The physical simulations are based on the work by Jakobsen [16]. 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" [17].

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.

It is very easy to begin developing with D3. The API is described in detail in the D3 Wiki [8], and the D3 website[3] includes an extensive array of examples that new developers can use as a starting off point. The D3 developer community is very large, so it is easy to find answers to almost every question online.

Listing 1: Dynamically create a dropdown menu.

```
// Select element with id "body" and append a select tag onto it. When it
    changes, the defined function will be triggered.
var dropdown = d3.select("#body")
            .append("select")
            .on("change", function() {
              var choice = this.options[this.selectedIndex].__data__;
              // handle choice
            });

var options = [{id: 1, name: "Option 1"},
         {id: 2, name: "Option 2"},
         {id: 3, name: "Option 3"}];

// Create an option tag with id and text attributes for each option that is
     defined in options
dropdown.selectAll("option")
    .data(options)
    .enter()
    .append("option")
    .attr("id", function(d) { return "op" + d.id; })
    .text( function(d) { return d.name; });
```

[8] https://github.com/mbostock/d3/wiki

```
// Select option 3 by default
dropdown.select("#op3")
    .attr("selected", true);
```

Listing 2: Html generated from Listing 1

```
<div id=body>
    <select>
        <option id="op1">Option 1</option>
        <option id="op2">Option 2</option>
        <option id="op3" selected=true>Option 3</option>
    </select>
</div>
```

3.3 GraphViz integration with emscripten

The current visualizations available the Tcl/Tk version of ProB are powered using the GraphViz[1] graph visualization software. There is support for generating graphs for the state space in the DOT graph language. The problem with this, and the reason that we are researching other alternatives, is that drawing GraphViz graphs is rather inefficient. However, for the state spaces that are derived using the signature merge algorithm, or for the transition diagrams that can be created from a state space, these graphs are quite pleasing to the eye. The derived graphs are also usually small enough that they can be drawn efficiently.

For this reason, we wanted to somehow be able to visualize small graphs written in the DOT language. In order to do this, we took advantage of the emscripten compiler [18]. This is a compiler that compiles LLVM bitcode to JavaScript so that it can be run in any modern browser. C programs can be compiled to LLVM. We used the Viz.js JavaScript library[9] developed by Mike Daines who has used emscripten in order to compile GraphViz from C to JavaScript and has provided a wrapper function to produce SVG visualizations. For example, the code shown in Listing 3 will produce Figure 1.

Listing 3: Create a visualization with viz.js and insert it into an html page.

```
svg = Viz("digraph { a -> b; a -> c; }", "svg");
$("#elementId").replaceWith(svg);
```

[9] https://github.com/mdaines/viz.js

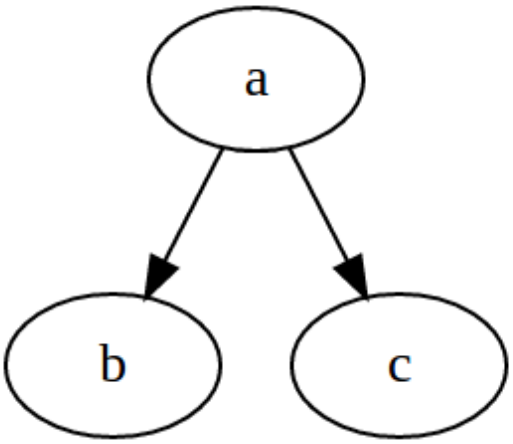


Figure 1: Generated graph image from Listing 3.

4 Contribution

4.1 Visualization framework

4.1.1 Structure

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 ProB 2.0. A functioning jetty server was already available, so we had to integrate our JavaScript visualizations into the existing framework. We ran into problems at the beginning, because a Java Servlet[10] is a singleton object. Ideally, we would have a unique servlet for every visualization because each visualization has its own set of data that needs to be visualized. In order to solve this problem, we created a session based servlet. When the user wants to create a new visualization, a unique session id is created. This is sent to the session based servlet, which creates a visualization engine responsible for the data calculation. When the visualization is initialized, it sets up a polling interval to ask the servlet for either the full dataset or for the changes since the last poll (see Figure 2). When the visualization polls the servlet, it includes its unique session id. The servlet then forwards the request to the visualization engine responsible for the data calculations which then returns the correct dataset.

The servlet is also connected to ProB 2.0 through the listener framework. Depending on the data that is needed for a particular visualization, the servlet registers either to receive notifications about changes in the current animation or about changes in the state space (see Figure 3). Every time the servlet receives a new notification, the data needed for the visualization is recalculated.

[10] http://docs.oracle.com/javaee/6/api/javax/servlet/Servlet.html

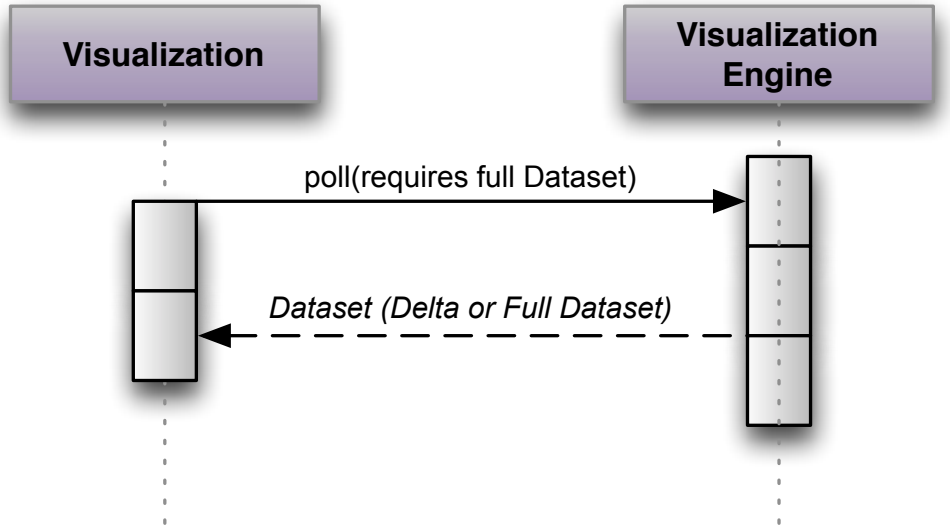


Figure 2: Visualizations poll the servlet to receive the dataset that needs to be visualized.

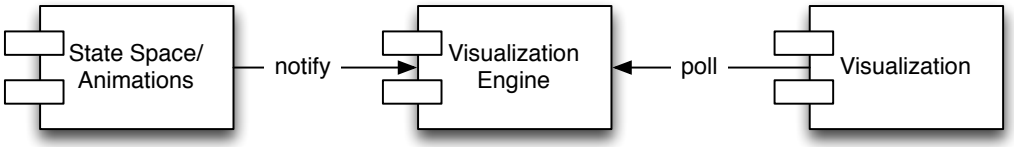


Figure 3: Model of the program flow.

The visualizations also support some persistence. For each visualization engine that is created, the settings for the visualization are saved with the given session id. If, over the course of the user's interaction with the visualization, the settings for the visualization change, the cached settings are updated. For example, if the user is visualizing the state space and has chosen to create a transition diagram for a given expression, both the expression and the type of graph that the user is viewing are saved. When the Eclipse application closes, the URL for each visualization that is open is saved (e.g. `http://localhost:8080/statespace_servlet/?init=`*`sessionId`*). The next time that the application is started, the cached settings for the given session id will be retrieved, and the visualization will be restored from the settings.

4.1.2 User customization of the visualizations

We wanted the user to be able to customize the appearance of the visualizations. One of the main advantages of the D3 visualization framework is the flexibility that it provides for the developer. Using D3 selectors, it is possible for a developer to select and change

the attributes of any of the elements of the visualization. We wanted the user to also be able to manipulate the visualization from within ProB.

In order to allow the user to directly manipulate the DOM of the visualization and thereby customize the appearance of the visualization, we decided to lift the functionality of the D3 selectors from the JavaScript level into the Java application. For this purpose, we defined a `Transformer` object. The user can create a `Transform` by defining a selector string based on the W3C Selectors API[11] and by calling the `set` method. The `set` method saves the name of the attribute that should be changed and the value to which the specified attribute should be changed. Because the `set` method returns the instance of the `Transformer`, it is possible chain the method calls together (see Lisiting 4).

Listing 4: Define a `Transformer` in Java

```
// Select elements with ids "sroot" and "s1" and set their fill and stroke
    attributes
Transformer t = new Transformer("#sroot,#s1").set("fill","red").set("stroke
    ","gray");
```

Every visualization engine contains a list of `Transformers` that is sent to the visualization every time that it is polled. The visualization can then read the selector and the attributes saved in the `Transformer` and apply the desired actions to the DOM. We now needed a way for the user to create `Transformers` and add them to the visualization. It is possible to add a `Transformer` to a visualization by calling the `apply` method in the visualization engine responsible for the visualization. When a new visualization engine is created, a variable with a reference to the object is loaded into the Groovy execution engine so that the user can access the visualization engine from within the Groovy REPL. We have also made the method `transform` available in the Groovy REPL so that the user can easily create `Transformers` (see Listing 5).

Listing 5: Create and apply a `Transformer` in the Groovy REPL

```
// Select elements with ids "sroot" and "s1" and set their fill and stroke
    attributes
x = transform("#sroot,#s1") {
        set "fill", "red"
        set "stroke", "gray"
}

// Apply to visualization
viz0.apply(x)
```

It is also possible to harness the power of the Groovy closure in order to create a Transformer that can be parameterized (see Listing 6).

Listing 6: Use Groovy closures to generate `Transformers`

```
// Create a closure that can be parameterized
colorize = { selection, color ->
                transform(selection) {
                    set "fill", color
```

[11] http://www.w3.org/TR/selectors-api/

```
                }
            }

// Color elements "sroot" and "s1" green
viz0.apply(colorize("#sroot,#s1", "green"))
```

The downside to this mechanism is that the user needs to have a good idea about the internal representation of the DOM in order to manipulate the visualizations. This mechanism is especially interesting for the state space visualization, so we spent more time defining the ids for the DOM elements. The SVG objects that are created using D3 are generated using the state ids and transition ids that are assigned by ProB. The mechanism for the definition of these ids can be seen in Table 1.

Table 1: Generated element ids for elements in the state space visualization.

Element	**Generated Element Id**
transition	`t` + *transition id*
text on transition	`tt` + *transition id*
state	`s` + *state id*
text on state	`st` + *state id*

It is also possible in ProB to filter a state space based on a predicate. This means that it is possible to give ProB a predicate and receive a list of state ids for which the given predicate holds. Ideally, however, we would want a `Transformer` based on a given predicate to be updated in the case that the state space changes.. For this purpose, we have introduced a `Transformer` that will be updated in the case that new states are added to the state space. The user can create such a `Transformer` in the Groovy REPL with the `transform` method by specifying a predicate and a state space object (see Listing 7). Currently, this will only update the state objects in the DOM, but in the future, we will be able to apply the same concept for all the different elements in the DOM.

Listing 7: Create a `Transformer` based on the states that match a given predicate

```
// Define your formula
predicate = "active\\/waiting={}" as ClassicalB

// Create a Transformer that will filter the given state space (saved in
    space0) according to the given predicate
x = transform(predicate, space_0 ) {
    set "fill", "blue"
    set "stroke", "white"
}

// Apply the Transformer to the visualization
viz0.apply(x)
```

4.1.3 Extensibility

In addition to creating visualizations that are useful to the user, we also wanted to make it easy for other developers to create similar visualizations. In order to help with this, we encapsuled certain elements common to all of the visualizations into a separate script. In order to have access to these elements, a developer simply needs to include the script before that of his visualization. Currently, there are two main elements included in this script. Firstly, the user can use the `createCanvas` function to create a D3 selection that includes support for zooming (see Listing 8). Secondly, if the user wants to be able to apply the `Transformers` that are described in the last section, there is a built in function to apply a list of `Transformers` to the visualization (see Listing 9). In the future, it will also be possible to add more functions to this script to make it even easier to create visualizations for ProB.

Listing 8: Append a D3 selection to an element that includes support for zooming

```
var width = 600, height = 400;
var svg = createCanvas("#elementId", width, height);

// Append all further elements for the visualization to this selection
svg.append(...);
```

Listing 9: Apply list of `Transformers` received from servlet

```
// The servlet has been polled, and a response has been received
var styling = response.styling;

// ... Render the visualization
// ... Then apply the user defined styling
applyStyling(styling);
```

4.2 Visualization of the State Space

4.2.1 Main visualization

When a state space visualization is opened, the visualization engine responsible for the state space visualization takes the state space associated with the current animation and extracts the information about the states and transitions contained within the graph. This information is then processed using the D3 force layout and rendered to create a visualization (see Figure 4). As a basis for this visualization, we used the Force Directed Graph example created by Michael Bostock (see Appendix A). A user can open the state space visualization by selecting Analyze > Open State Space Visualization in the ProB menu of the ProB 2.0 application. This will open a new visualization of the state space for the current animation.

Unfortunately, in some cases the state space contains an extremely large amount of information that has to be processed. This includes the values of the variables and the invariant for every state in the graph and the names and parameters of the transitions that correspond to every edge in the graph. Because of this, it is rather difficult to create

a useful visualization of the whole state space because the user not only wants to inspect the structure of the state space as a whole but also the individual states.

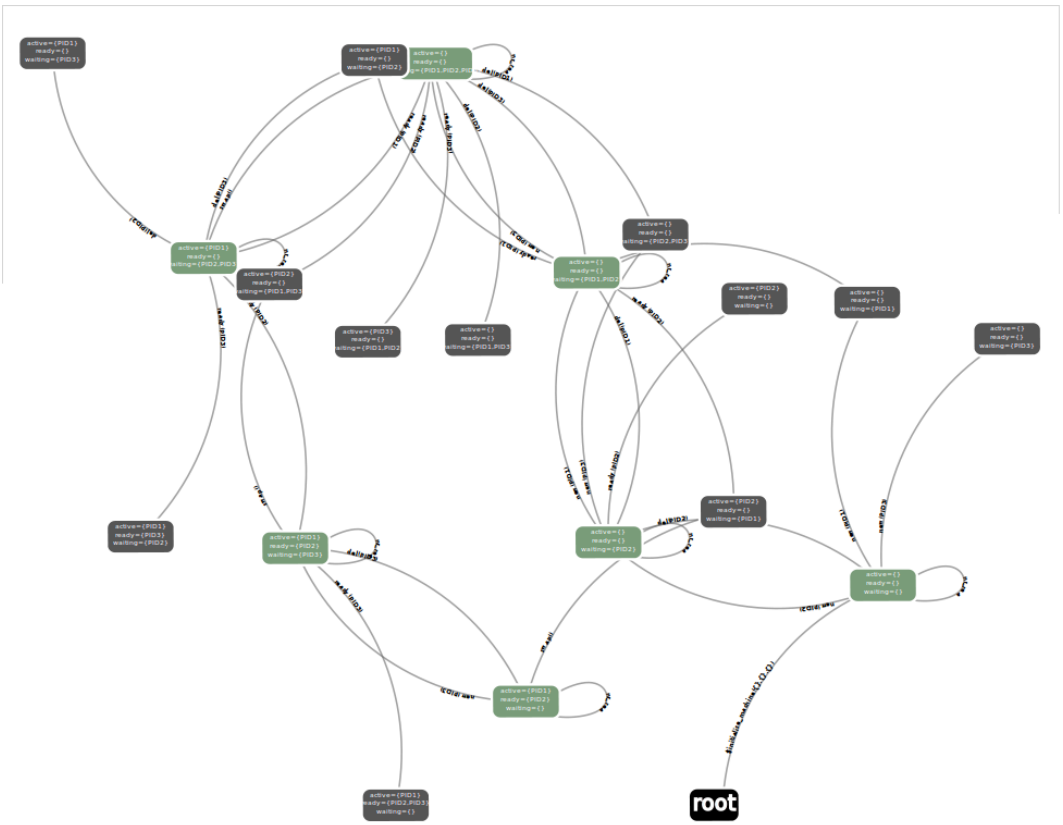


Figure 4: Visualization of a partially explored state space for the Scheduler example

As a proposed solution of 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 state and transition objects very small. When the visualization is created, the user can view the graph to see how the state space appears as a whole. The text for the individual states, however, is virtually indiscernable. If the user wants to inspect a particular state, they can do so by zooming into the visualization. The text is then larger, and the user can see the values of the variables for the given state and the outgoing transitions (see Figure 5). Then user can also click on the background of the visualization in order to pan through the visualization and inspect other states and transitions. The visualization is also interactive. The user can grab a state and move it around to a desired position.

We had some performance issues that were associated with the rendering very large state spaces. The force layout keeps adjusting the graph until it reaches a fixpoint. The problem was that as the state space grew, there were more and more objects that had to be rendered. The force calculates the position of the vertices in the graph iteratively. By default, the graph is rerendered in every iteration of the calculation of the graph layout. When the number of states in a state space is very large, the rendering and therefore the calculation of the layout, takes a lot of resources. This affected not only the appearance of the visualization, but it also cost enough resources that the whole eclipse plugin became unresponsive. The solution we found to the problem was to allow the user to turn off the rendering of the graph. The user can do this by selecting the play/pause button in the upper left hand corner of the visualization (see Figure 6). The visualization will then freeze in the position that is is currently drawn. The force layout, however, will continue running in the background, so if the user decides to turn on the rendering, the graph will hopefully have had time to stabilize.

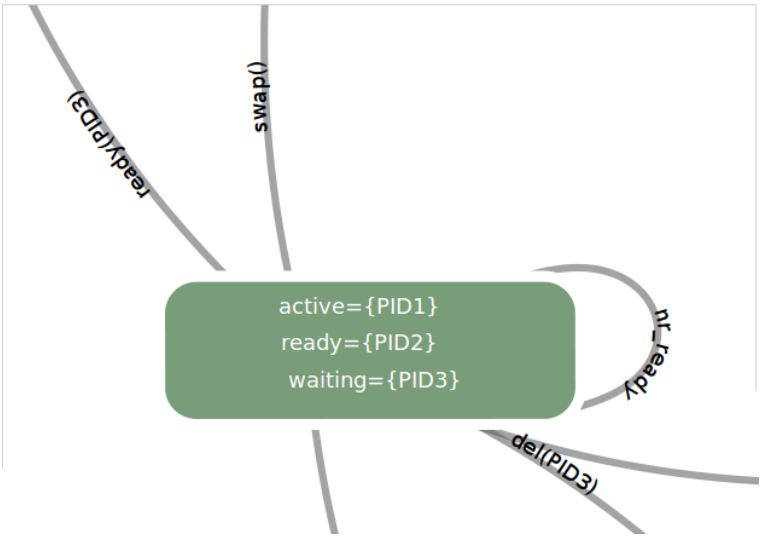


Figure 5: By zooming in, the user can inspect individual nodes.

When new states are added to the state space, these are added to the graph when the visualization receives them in the next poll. A node is inserted at the same position as the preceding state in the graph. The graph is then updated. This results in a nice animation because the new states will appear to pop outward from their parent state.

Status about the invariant is available in the graph based on the color of the nodes. If an invariant violation is present, the state is colored red. If the invariant is ok, the state is colored green. Otherwise, if the invariant has not yet been calculated for the given state, the node is colored gray. The labels for the states are the values of the variables for the given state.

The state spaces often grow exponentially. This is known as the state space explosion problem. For this reason, a visualization of the entire state space can be too much information for a user to process at any given time. In order to help understand the overall

behavior of the state space, we have also made smaller graphs available that are derived from the original state space and can be more useful for the user. The user can choose from these visualizations in the dropdown menu in the upper left hand corner (see Figure 6).

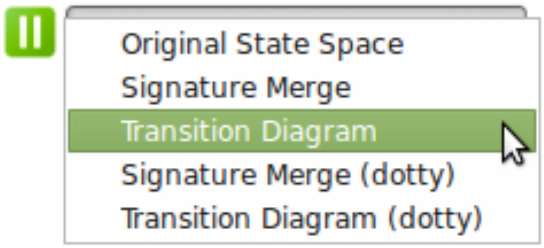


Figure 6: The user can select the desired visualization and play and pause the rendering.

4.2.2 Signature Merged State Spaces

The signature merge algorithm is the first of two algorithms for reducing the state space that have been implemented in this work (see Figure 7).

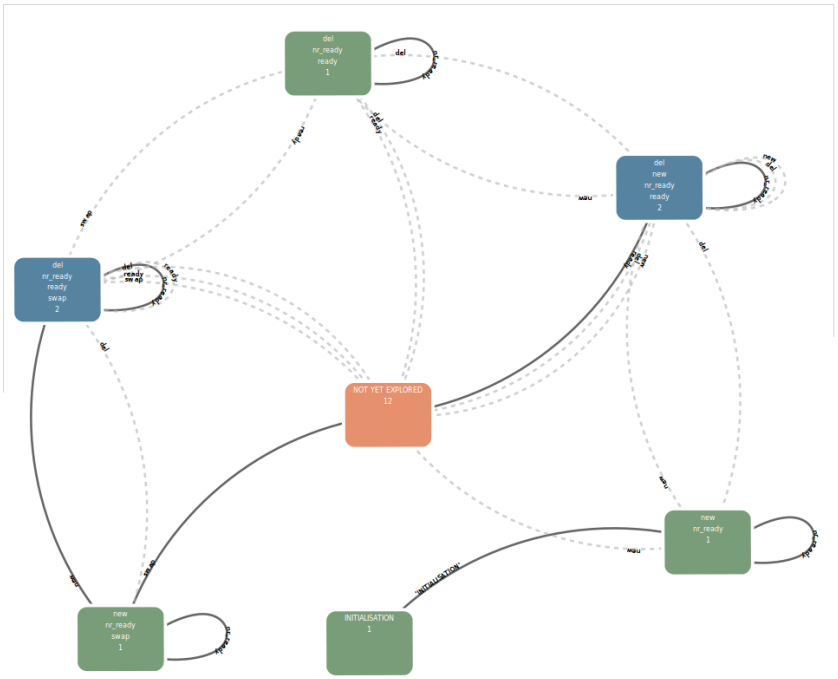


Figure 7: D3 Visualization of signature merge for the Scheduler example

The algorithm works by merging all of the states which have the same outgoing transitions. This creates a state space that is considerably smaller but that still preserves information about the operations that are enabled for a given state.

If the user wants to use the GraphViz algorithms and rendering engine, there is also support for visualizing the DOT representation of the signature merged state space that is generated from ProB (see Figure 8).

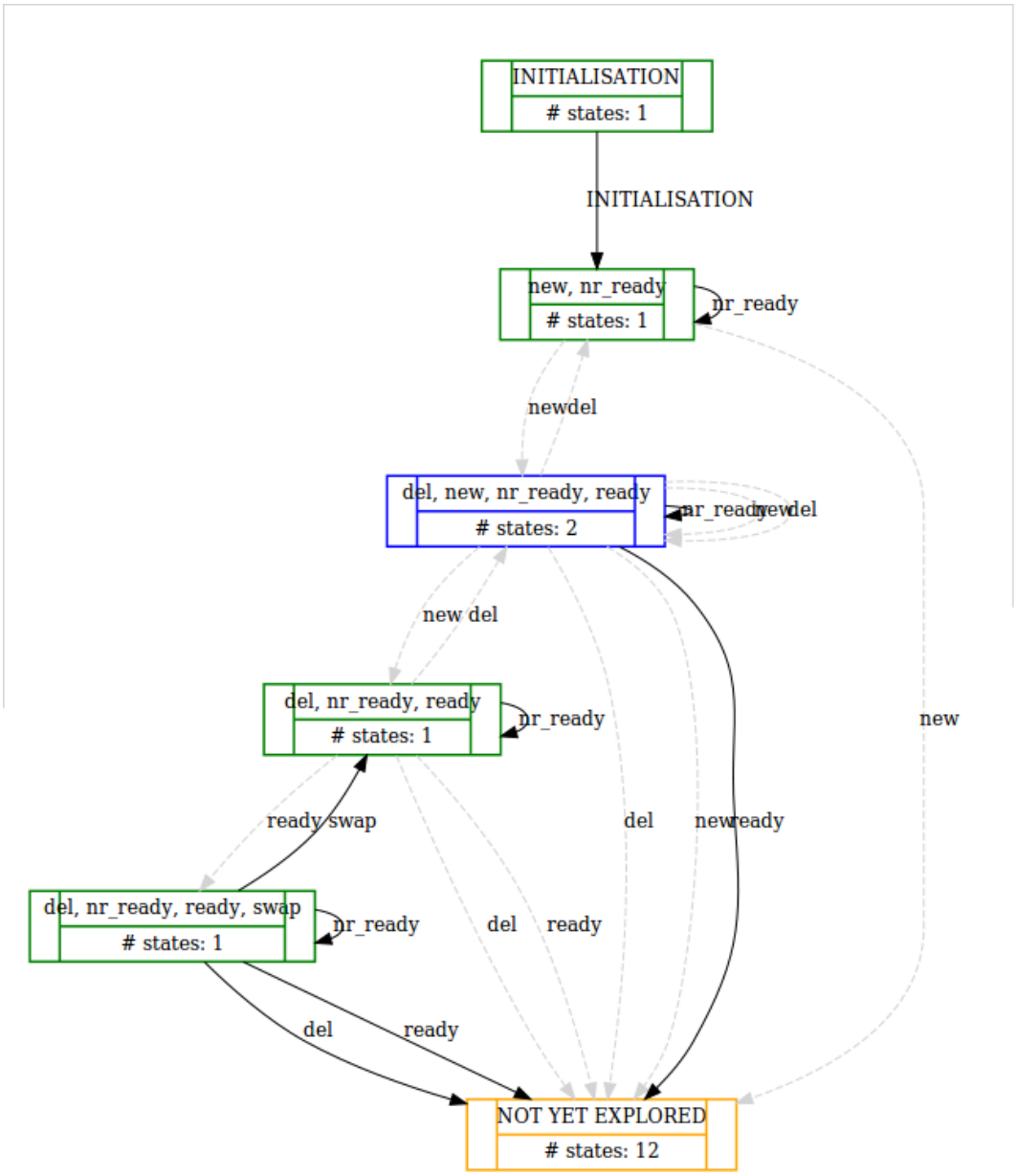


Figure 8: GraphViz powered visualization of the signature merge algorithm for the Scheduler example

Rendering of the DOT graph specification is done using the Viz.js JavaScript library. The visualization supports zooming and panning in the same way as the D3 powered visualizations do.

The graph that is generated with the signature merge algorithm differs based on the transitions that are of the interest of to the user. By default, the algorithm is calculated using all of the transitions that are present in the model. However, it also possible to select the desired transitions for the calculation of the algorithm. In order to allow the user to select the transitions of interest, we have implemented a small user interface that pops up when the user clicks on the settings icon (see Figure 9). The settings icon only becomes available when the user is in a mode dealing with the signature merge algorithm. The label for the nodes in this graph consists of a list of the outgoing transitions from the node and the number of states that have been merged into the node.

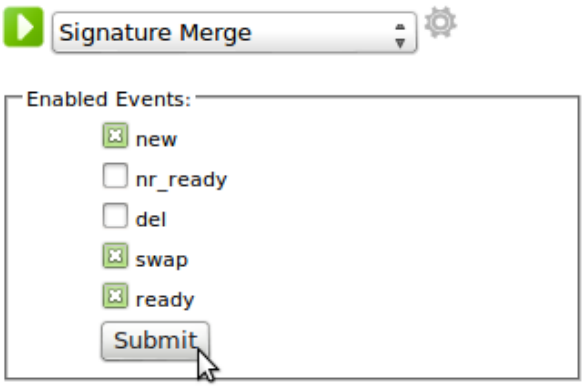


Figure 9: User interface to chose events for signature merge.

4.2.3 Transition Diagrams

The other reduction algorithm that is supported in this implementation is the creation of transition diagrams. In order to perform this algorithm, ProB receives an expression from the user. Then ProB calculates all of the possible solutions to the expression in the scope of the model that is being animated. These become the vertices in the graph. The edges in the graph show the transitions that change the value of the given expression to that of the value shown in the target state (see Figure 11).

If the user prefers the GraphViz represenation over the D3 powered visualization, it is also possible to generate a GraphViz based visualization for the transition diagram (see Figure 12). As with the signature merged state space, the UI for the GraphViz powered transition diagram is the same as that for the D3 powered visualization, and the visualization supports zooming and panning. Although the GraphViz algorithms are inherently

offline, these visualizations make use of the visualization framework and are recalculated and rerendered every time that a change takes place within the state space.

When the user chooses to create a transition diagram from the menu, a prompt appears. This is how the user can specify the initial expression for the calculation of the transition diagram. Once the graph is calculated and rendered, a text field appears next to the drop down menu (see Figure 10). If the user wants to change the expression that is being visualized, they can input a new expression here and submit it. The algorithm will then be recalculated and the graph will be rerendered.

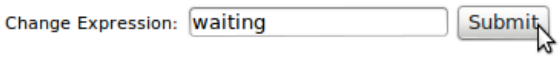


Figure 10: The user can input a new expression to recalculate the transition diagram.

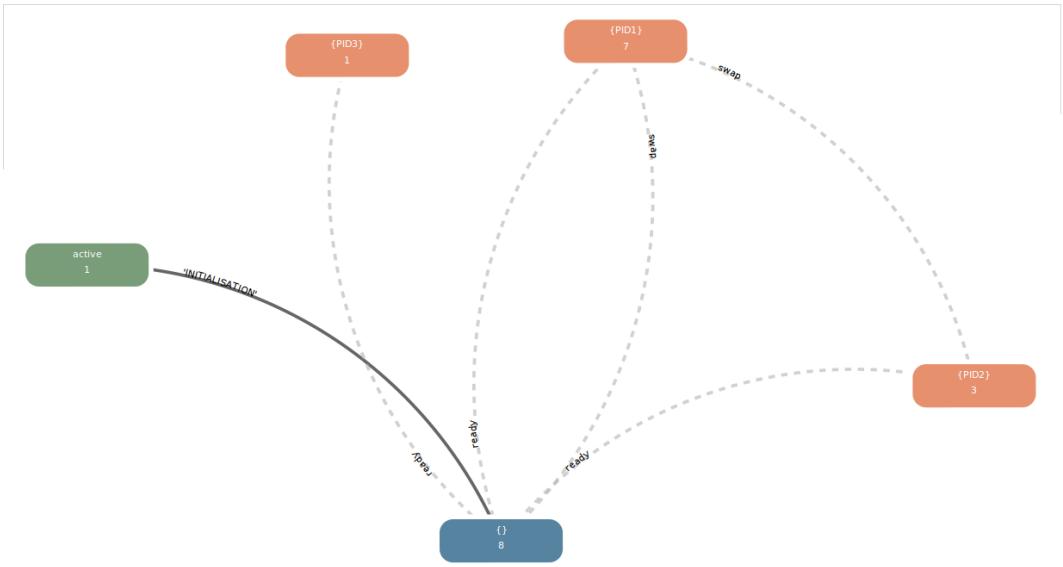


Figure 11: D3 Visualization of the transition diagram of `active` in the Scheduler example

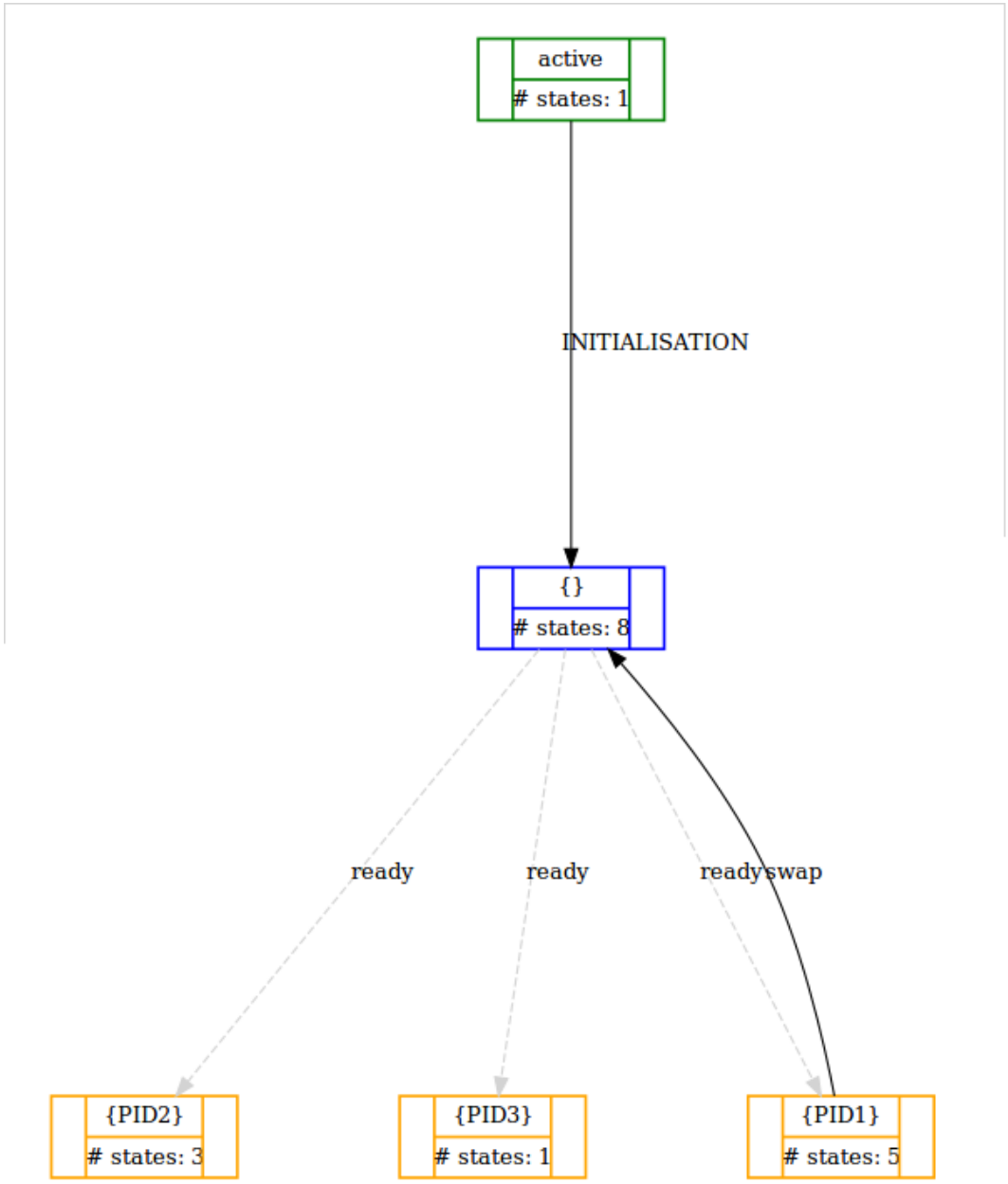


Figure 12: GraphViz powered visualization of the transition diagram of `active` in the Scheduler model

4.3 Visualization of a B-type Formula

ProB already supported the functionality of expanding a formula into its subformulas and finding its value at a given state. However, for any given formula, only the subformulas directly under the desired formula would be calculated. This algorithm was

adapted so that all the subformulas are calculated recursively and a tree structure is built automatically. This structure is then cached, and can then be evaluated for any given state. The user can create a new visualization for a formula by selecting Analyze > Open Visualization of Formula from the ProB menu and entering the desired formula in the prompt that pops up, or by right clicking on a formula in the State Inspector view and selecting Open Visualization of Formula.

The final visualization is interactive (see Figure 13). 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 on whether 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.

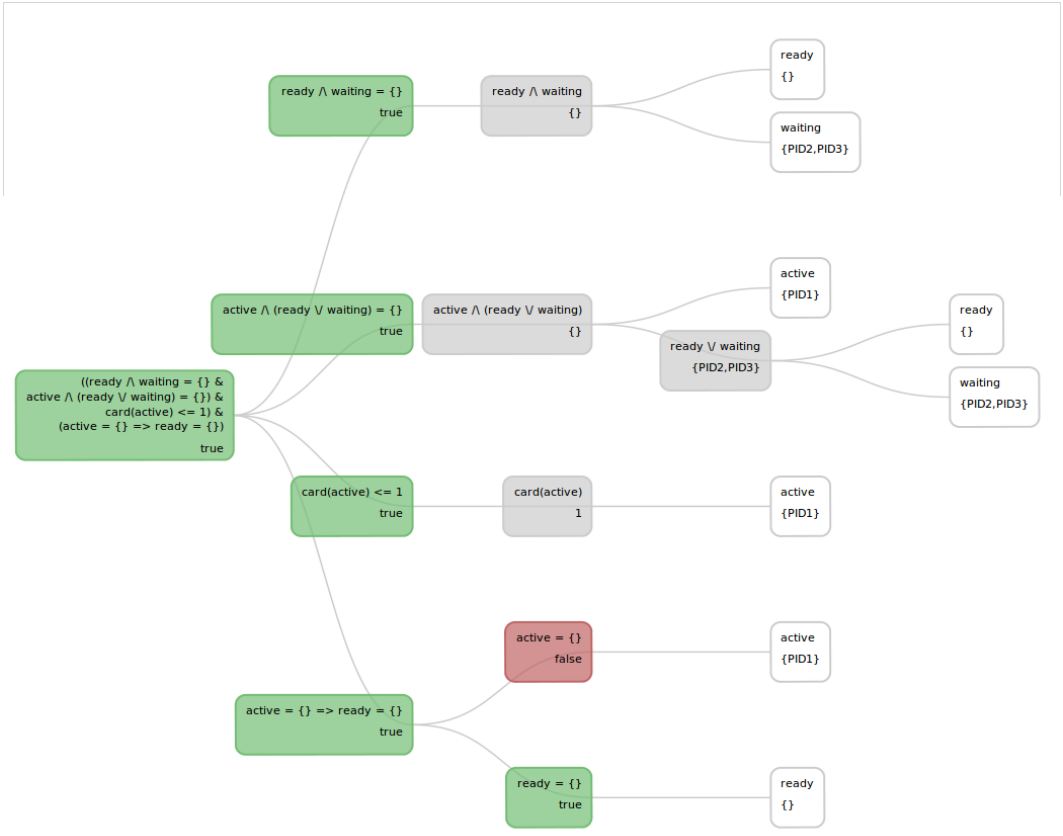


Figure 13: Visualization of the invariant of the Scheduler model

In this visualization, the D3 tree layout is used. The expanding and collapsing of the nodes takes place with a simple JavaScript function. This design of this visualization

is based on the Collapsible Tree Layout from the D3 website (see Appendix B). 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. When the current state in the animation changes, the formula is recalculated and the visualization is redrawn.

4.4 Visualization of the value of a formula over time

The last visualization that we have created in the course of this work is the visualization of the value of a given formula over the course of an animation. This can be especially useful if the user is interested in analyzing models with a temporal element.

To create this visualization, the user can select Analyze > Open Time vs Value Visualization from the ProB menu. A prompt will then pop up and the user can specify the formula that is to be evaluated for all the states in the current animation. Optionally, the user can also specify a second expression which will represent the temporal element for the model. In this case, the values of the first formula will be plotted against the values of the expression that the user has inputted. Otherwise, the values of the first formula will be plotted against the number of animation steps.

The value pairs that are produced through this evaluation are processed by D3 to produce a simple line plot (see Figure 14). If the current state changes, the formula is recalculated and a new plot is produced.

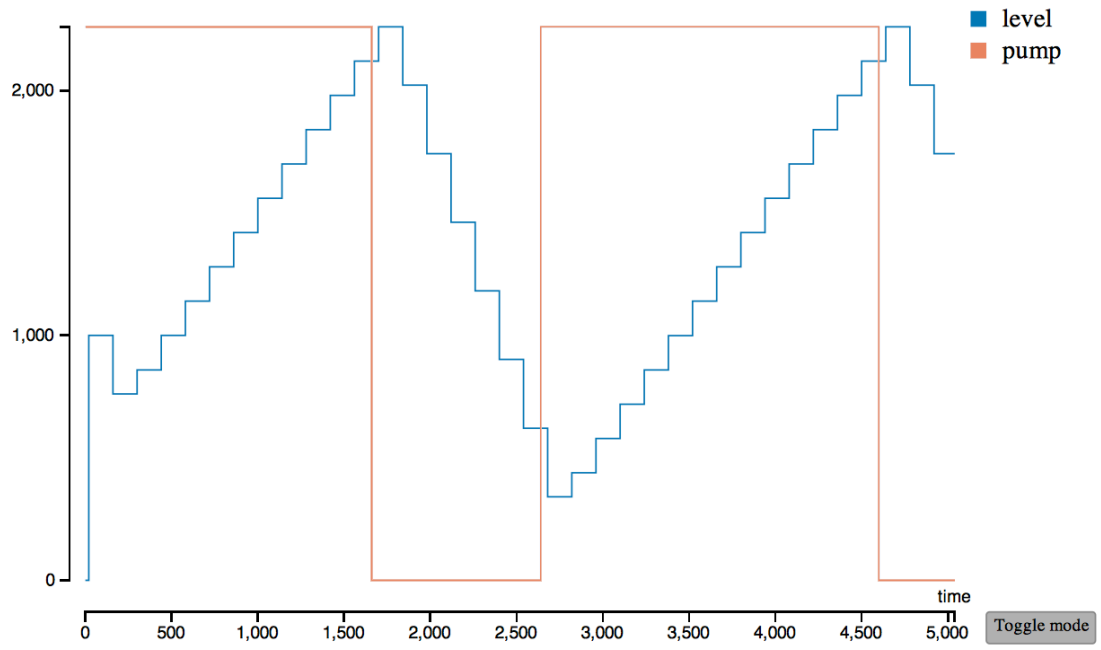


Figure 14: Visualization of the value of `level` and `pump` over `time` for the water tank example.

It is possible to visualize multiple formulas at the same time. In order to add a formula to the visualization, the user can input a new formula in the text box in the upper left hand corner (see Figure 15).

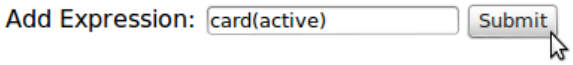


Figure 15: The user can add an expression to the visualization.

There are two modes for viewing the line plots. The first mode plots all of values in the same line plot. If the user want to view each individual line separately, he can select the Toggle Mode button in the lower right corner of the visualization. This will produce a separate line plot for each formula (see Figure 16).

It is possible to visualize predicates and expressions that take on integer or boolean values. If the formula that is being visualized is a predicate or an expression that takes on a boolean value, the resulting value is mapped to an integer value. If all of the formulas are being drawn in the same line plot, the boolean value TRUE will be mapped to the maximum value of all the data sets. Otherwise, it will be mapped to 1. The boolean value FALSE is always mapped to 0.

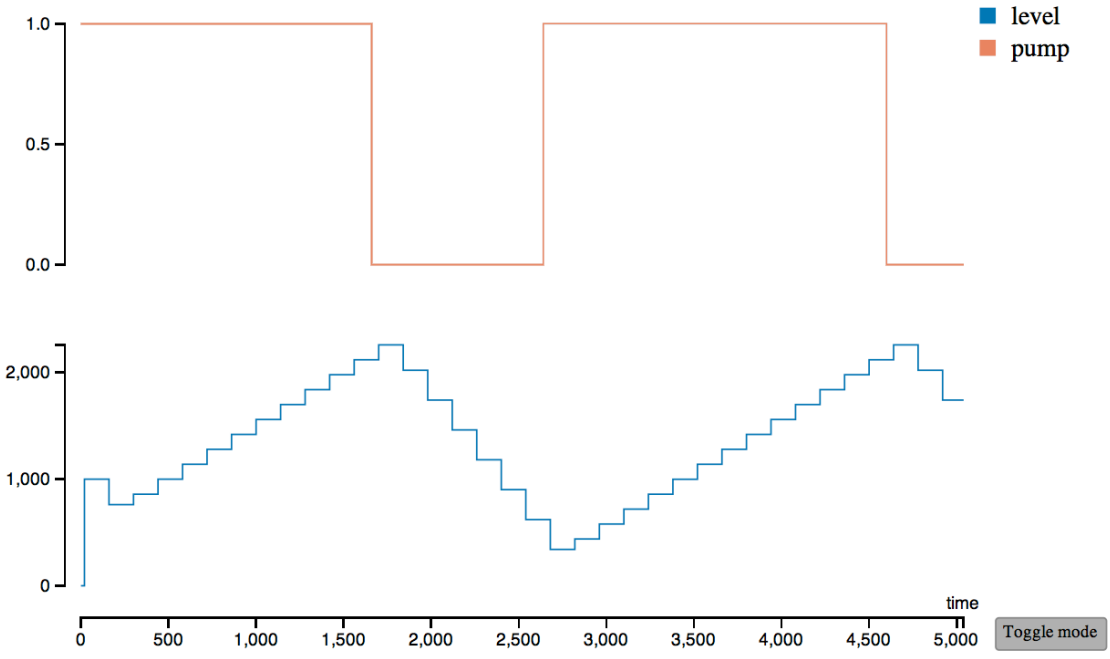


Figure 16: Visualization of separate line charts for each inputted formula in the water tank example.

5 Future Work

One of the main features we implemented in this work was the creation of a visualization framework. We have attempted to make the elements used for the visualization engines and the visualizations extensible. It should now be relatively simple to create new visualizations and integrate them into the existing visualization framework.

In the future, we will use this basis to create other useful visualizations for the user. ProB 2.0 still does not have all of the visualizations that are available in the Tcl/Tk version of ProB. For instance, we still need to implement the graphical visualization of a state within ProB 2.0. We will also likely create a visualization of the refinement hierarchy that is present in Classical B and Event-B models.

One of the projects in ProB 2.0 that is currently underway is the development of a worksheet element. This will serve as documentation and a means to run groovy scripts within the application. In the future, it should be possible to embed the visualizations created in the scope of this work within the worksheet element.

Because time was limited during the course of this work, we have not had the time to implement all of the features that we would have liked to implement for all of the visualizaitons. Future features for the existing visualizations could include:

- Support for the visualization of formulas in specification languages other than Classical B and Event-B.
- Redefinition of the element ids for the Formula and Value Over Time visualizations so that it is easier for the user to customize the appearance of the visualizations.
- Introduction of a better mechanism to manage (i.e. add and remove) formulas from the Value Over Time visualization.

6 Conclusion

During the course of this work, we were able to fulfill all of our objectives. We were able to create three new data visualizations that will be useful for users of ProB. These visualizations are generated dynamically from data that is produced by ProB during consistency checking and the animation of formal specifications. They are updated automatically whenever any changes in the state space or in the current animation take place. We have also created a way for the user to customize the appearance of the visualizations.

The visualization framework that we have created is extensible. It should be possible to create new data visualization that will fit into the existing framework. It should also be possible to adapt the existing visualizations to add functionality. For instance, the decision to support GraphViz visualizations in the state space visualization took place relatively late in the development process. Using the existing visualization framework, it was possible to implement this feature in a relatively short period of time. This shows that the visualization framework is relatively flexible and can be easily extended.

We have shown the feasibility of using D3 to create data visualizations within ProB. By using the browser widgets available in the Eclipse RCP application, it was simple to integrate the JavaScript visualizations into ProB 2.0. These views appear to be native to the Eclipse platform, but there is a much higher level of user interaction possible with web applications than there is with native Eclipse views. For example, the zooming and panning capability available in all of the visualizations is a pure JavaScript function that is made available from D3.

All in all, adding data generated visualizations into the ProB 2.0 application has made the program more useful for the users. It is now not only possible for the user to extract and analyze textual data from ProB, but also to view and interact with a visualization based on the data.

References

[1] Michael Leuschel and Michael Butler. ProB: A model checker for B. In Araki Keijiro, Stefania Gnesi, and Mandrio Dino, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer-Verlag, 2003.

[2] J.R. Abrial, J.R. Abrial, and A. Hoare. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[3] S. Schneider. *The B-Method: An Introduction*. Cornerstones of Computing Series. Palgrave Macmillan Limited, 2001.

[4] Lukas Ladenberger, Jens Bendisposto, and Michael Leuschel. Visualising Event-B models with B-Motion Studio. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Proceedings of FMICS 2009*, volume 5825 of *Lecture Notes in Computer Science*, pages 202–204. Springer, 2009.

[5] Michael Leuschel and Edd Turner. Visualising larger state spaces in ProB. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 6–23. Springer-Verlag, November 2005.

[6] Michael Leuschel, Mireille Samia, Jens Bendisposto, and Li Luo. Easy graphical animation and formula viewing for teaching B. In C. Attiogbé and H. Habrias, editors, *The B Method: from Research to Teaching*, pages 17–32. Lina, 2008.

[7] Marc Fontaine. *A Model Checker for CSP-M*. PhD thesis, Heinrich Heine Universität Düsseldorf, July 2011.

[8] Frank van Ham, Huub van de Wetering, and Jarke J. van Wijk. Interactive visualization of state transition systems. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, 8(3):2002, 2002.

[9] A. Johannes Pretorius and Jarke J. van Wijk. Multidimensional visualization of transition systems. In *Proceedings of the Ninth International Conference on Information Visualisation*, IV '05, pages 323–328, Washington, DC, USA, 2005. IEEE Computer Society.

[10] A. Johannes Pretorius and Jarke J. Van Wijk. Visual analysis of multivariate state transition graphs. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):685–692, September 2006.

[11] Michael Leuschel and Michael Butler. ProB: An automated analysis toolset for the B method. *Software Tools for Technology Transfer (STTT)*, 10(2):185–203, 2008.

[12] Michael Butler and Stefan Hallerstede. The Rodin formal modelling tool. In *BCS-FACS Christmas 2007 Meeting*, 2007.

[13] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[14] Edward M. Reingold, John, and S. Tilford. Tidier drawing of trees. *IEEE Trans. Software Eng*, 1981.

[15] Tim Dwyer. Scalable, versatile and simple constrained graph layout. In *Proceedings of the 11th Eurographics / IEEE - VGTC conference on Visualization*, EuroVis'09, pages 991–1006, Aire-la-Ville, Switzerland, Switzerland, 2009. Eurographics Association.

[16] Thomas Jakobsen. Advanced Character Physics, 2003.

[17] Michael Bostock. D3 wiki. `https://github.com/mbostock/d3/wiki/Force-Layout`, 2012. Accessed: 2013-05-27.

[18] Alon Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 301–312, New York, NY, USA, 2011. ACM.

List of Figures

List of Tables

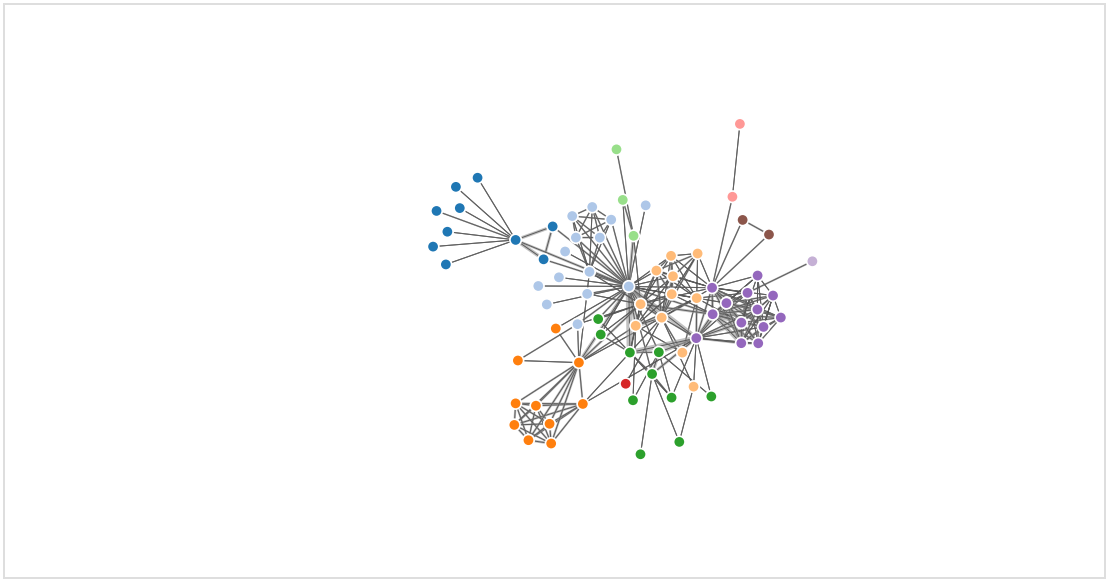
Listings

A Force Directed Graph

mbostock's block #4062045

Force-Directed Graph

May 8, 2013



This simple force-directed graph shows character co-occurence in *Les Misérables*. A physical simulation of charged particles and springs places related characters in closer proximity, while unrelated characters are farther apart. Layout algorithm inspired by Tim Dwyer and Thomas Jakobsen. Data based on character coappearence in Victor Hugo's *Les Misérables*, compiled by Donald Knuth.

Open in a new window.

index.html

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>

.node {
  stroke: #fff;
  stroke-width: 1.5px;
}

.link {
  stroke: #999;
  stroke-opacity: .6;
}

</style>
<body>
<script src="http://d3js.org/d3.v3.min.js"></script>
<script>

var width = 960,
    height = 500;
```

```
var color = d3.scale.category20();

var force = d3.layout.force()
    .charge(-120)
    .linkDistance(30)
    .size([width, height]);

var svg = d3.select("body").append("svg")
    .attr("width", width)
    .attr("height", height);

d3.json("miserables.json", function(error, graph) {
  force
      .nodes(graph.nodes)
      .links(graph.links)
      .start();

  var link = svg.selectAll(".link")
      .data(graph.links)
    .enter().append("line")
      .attr("class", "link")
      .style("stroke-width", function(d) { return Math.sqrt(d.value); });

  var node = svg.selectAll(".node")
      .data(graph.nodes)
    .enter().append("circle")
      .attr("class", "node")
      .attr("r", 5)
      .style("fill", function(d) { return color(d.group); })
      .call(force.drag);

  node.append("title")
      .text(function(d) { return d.name; });

  force.on("tick", function() {
    link.attr("x1", function(d) { return d.source.x; })
        .attr("y1", function(d) { return d.source.y; })
        .attr("x2", function(d) { return d.target.x; })
        .attr("y2", function(d) { return d.target.y; });

    node.attr("cx", function(d) { return d.x; })
        .attr("cy", function(d) { return d.y; });
  });
});

</script>
```

miserables.json

```
{
  "nodes":[
    {"name":"Myriel","group":1},
    {"name":"Napoleon","group":1},
    {"name":"Mlle.Baptistine","group":1},
    {"name":"Mme.Magloire","group":1},
    {"name":"CountessdeLo","group":1},
    {"name":"Geborand","group":1},
    {"name":"Champtercier","group":1},
    {"name":"Cravatte","group":1},
    {"name":"Count","group":1},
    {"name":"OldMan","group":1},
    {"name":"Labarre","group":2},
    {"name":"Valjean","group":2},
    {"name":"Marguerite","group":3},
    {"name":"Mme.deR","group":2},
```

```
    {"name":"Isabeau","group":2},
    {"name":"Gervais","group":2},
    {"name":"Tholomyes","group":3},
    {"name":"Listolier","group":3},
    {"name":"Fameuil","group":3},
    {"name":"Blacheville","group":3},
    {"name":"Favourite","group":3},
    {"name":"Dahlia","group":3},
    {"name":"Zephine","group":3},
    {"name":"Fantine","group":3},
    {"name":"Mme.Thenardier","group":4},
    {"name":"Thenardier","group":4},
    {"name":"Cosette","group":5},
    {"name":"Javert","group":4},
    {"name":"Fauchelevent","group":0},
    {"name":"Bamatabois","group":2},
    {"name":"Perpetue","group":3},
    {"name":"Simplice","group":2},
    {"name":"Scaufflaire","group":2},
    {"name":"Woman1","group":2},
    {"name":"Judge","group":2},
    {"name":"Champmathieu","group":2},
    {"name":"Brevet","group":2},
    {"name":"Chenildieu","group":2},
    {"name":"Cochepaille","group":2},
    {"name":"Pontmercy","group":4},
    {"name":"Boulatruelle","group":6},
    {"name":"Eponine","group":4},
    {"name":"Anzelma","group":4},
    {"name":"Woman2","group":5},
    {"name":"MotherInnocent","group":0},
    {"name":"Gribier","group":0},
    {"name":"Jondrette","group":7},
    {"name":"Mme.Burgon","group":7},
    {"name":"Gavroche","group":8},
    {"name":"Gillenormand","group":5},
    {"name":"Magnon","group":5},
    {"name":"Mlle.Gillenormand","group":5},
    {"name":"Mme.Pontmercy","group":5},
    {"name":"Mlle.Vaubois","group":5},
    {"name":"Lt.Gillenormand","group":5},
    {"name":"Marius","group":8},
    {"name":"BaronessT","group":5},
    {"name":"Mabeuf","group":8},
    {"name":"Enjolras","group":8},
    {"name":"Combeferre","group":8},
    {"name":"Prouvaire","group":8},
    {"name":"Feuilly","group":8},
    {"name":"Courfeyrac","group":8},
    {"name":"Bahorel","group":8},
    {"name":"Bossuet","group":8},
    {"name":"Joly","group":8},
    {"name":"Grantaire","group":8},
    {"name":"MotherPlutarch","group":9},
    {"name":"Gueulemer","group":4},
    {"name":"Babet","group":4},
    {"name":"Claquesous","group":4},
    {"name":"Montparnasse","group":4},
    {"name":"Toussaint","group":5},
    {"name":"Child1","group":10},
    {"name":"Child2","group":10},
    {"name":"Brujon","group":4},
    {"name":"Mme.Hucheloup","group":8}
  ],
  "links":[
    {"source":1,"target":0,"value":1},
    {"source":2,"target":0,"value":8},
    {"source":3,"target":0,"value":10},
```

```
    {"source":3,"target":2,"value":6},
    {"source":4,"target":0,"value":1},
    {"source":5,"target":0,"value":1},
    {"source":6,"target":0,"value":1},
    {"source":7,"target":0,"value":1},
    {"source":8,"target":0,"value":2},
    {"source":9,"target":0,"value":1},
    {"source":11,"target":10,"value":1},
    {"source":11,"target":3,"value":3},
    {"source":11,"target":2,"value":3},
    {"source":11,"target":0,"value":5},
    {"source":12,"target":11,"value":1},
    {"source":13,"target":11,"value":1},
    {"source":14,"target":11,"value":1},
    {"source":15,"target":11,"value":1},
    {"source":17,"target":16,"value":4},
    {"source":18,"target":16,"value":4},
    {"source":18,"target":17,"value":4},
    {"source":19,"target":16,"value":4},
    {"source":19,"target":17,"value":4},
    {"source":19,"target":18,"value":4},
    {"source":20,"target":16,"value":3},
    {"source":20,"target":17,"value":3},
    {"source":20,"target":18,"value":3},
    {"source":20,"target":19,"value":4},
    {"source":21,"target":16,"value":3},
    {"source":21,"target":17,"value":3},
    {"source":21,"target":18,"value":3},
    {"source":21,"target":19,"value":3},
    {"source":21,"target":20,"value":5},
    {"source":22,"target":16,"value":3},
    {"source":22,"target":17,"value":3},
    {"source":22,"target":18,"value":3},
    {"source":22,"target":19,"value":3},
    {"source":22,"target":20,"value":4},
    {"source":22,"target":21,"value":4},
    {"source":23,"target":16,"value":3},
    {"source":23,"target":17,"value":3},
    {"source":23,"target":18,"value":3},
    {"source":23,"target":19,"value":3},
    {"source":23,"target":20,"value":4},
    {"source":23,"target":21,"value":4},
    {"source":23,"target":22,"value":4},
    {"source":23,"target":12,"value":2},
    {"source":23,"target":11,"value":9},
    {"source":24,"target":23,"value":2},
    {"source":24,"target":11,"value":7},
    {"source":25,"target":24,"value":13},
    {"source":25,"target":23,"value":1},
    {"source":25,"target":11,"value":12},
    {"source":26,"target":24,"value":4},
    {"source":26,"target":11,"value":31},
    {"source":26,"target":16,"value":1},
    {"source":26,"target":25,"value":1},
    {"source":27,"target":11,"value":17},
    {"source":27,"target":23,"value":5},
    {"source":27,"target":25,"value":5},
    {"source":27,"target":24,"value":1},
    {"source":27,"target":26,"value":1},
    {"source":28,"target":11,"value":8},
    {"source":28,"target":27,"value":1},
    {"source":29,"target":23,"value":1},
    {"source":29,"target":27,"value":1},
    {"source":29,"target":11,"value":2},
    {"source":30,"target":23,"value":1},
    {"source":31,"target":30,"value":2},
    {"source":31,"target":11,"value":3},
    {"source":31,"target":23,"value":2},
```

```
    {"source":31,"target":27,"value":1},
    {"source":32,"target":11,"value":1},
    {"source":33,"target":11,"value":2},
    {"source":33,"target":27,"value":1},
    {"source":34,"target":11,"value":3},
    {"source":34,"target":29,"value":2},
    {"source":35,"target":11,"value":3},
    {"source":35,"target":34,"value":3},
    {"source":35,"target":29,"value":2},
    {"source":36,"target":34,"value":2},
    {"source":36,"target":35,"value":2},
    {"source":36,"target":11,"value":2},
    {"source":36,"target":29,"value":1},
    {"source":37,"target":34,"value":2},
    {"source":37,"target":35,"value":2},
    {"source":37,"target":36,"value":2},
    {"source":37,"target":11,"value":2},
    {"source":37,"target":29,"value":1},
    {"source":38,"target":34,"value":2},
    {"source":38,"target":35,"value":2},
    {"source":38,"target":36,"value":2},
    {"source":38,"target":37,"value":2},
    {"source":38,"target":11,"value":2},
    {"source":38,"target":29,"value":1},
    {"source":39,"target":25,"value":1},
    {"source":40,"target":25,"value":1},
    {"source":41,"target":24,"value":2},
    {"source":41,"target":25,"value":3},
    {"source":42,"target":41,"value":2},
    {"source":42,"target":25,"value":2},
    {"source":42,"target":24,"value":1},
    {"source":43,"target":11,"value":3},
    {"source":43,"target":26,"value":1},
    {"source":43,"target":27,"value":1},
    {"source":44,"target":28,"value":3},
    {"source":44,"target":11,"value":1},
    {"source":45,"target":28,"value":2},
    {"source":47,"target":46,"value":1},
    {"source":48,"target":47,"value":2},
    {"source":48,"target":25,"value":1},
    {"source":48,"target":27,"value":1},
    {"source":48,"target":11,"value":1},
    {"source":49,"target":26,"value":3},
    {"source":49,"target":11,"value":2},
    {"source":50,"target":49,"value":1},
    {"source":50,"target":24,"value":1},
    {"source":51,"target":49,"value":9},
    {"source":51,"target":26,"value":2},
    {"source":51,"target":11,"value":2},
    {"source":52,"target":51,"value":1},
    {"source":52,"target":39,"value":1},
    {"source":53,"target":51,"value":1},
    {"source":54,"target":51,"value":2},
    {"source":54,"target":49,"value":1},
    {"source":54,"target":26,"value":1},
    {"source":55,"target":51,"value":6},
    {"source":55,"target":49,"value":12},
    {"source":55,"target":39,"value":1},
    {"source":55,"target":54,"value":1},
    {"source":55,"target":26,"value":21},
    {"source":55,"target":11,"value":19},
    {"source":55,"target":16,"value":1},
    {"source":55,"target":25,"value":2},
    {"source":55,"target":41,"value":5},
    {"source":55,"target":48,"value":4},
    {"source":56,"target":49,"value":1},
    {"source":56,"target":55,"value":1},
    {"source":57,"target":55,"value":1},
```

```
    {"source":57,"target":41,"value":1},
    {"source":57,"target":48,"value":1},
    {"source":58,"target":55,"value":7},
    {"source":58,"target":48,"value":7},
    {"source":58,"target":27,"value":6},
    {"source":58,"target":57,"value":1},
    {"source":58,"target":11,"value":4},
    {"source":59,"target":58,"value":15},
    {"source":59,"target":55,"value":5},
    {"source":59,"target":48,"value":6},
    {"source":59,"target":57,"value":2},
    {"source":60,"target":48,"value":1},
    {"source":60,"target":58,"value":4},
    {"source":60,"target":59,"value":2},
    {"source":61,"target":48,"value":2},
    {"source":61,"target":58,"value":6},
    {"source":61,"target":60,"value":2},
    {"source":61,"target":59,"value":5},
    {"source":61,"target":57,"value":1},
    {"source":61,"target":55,"value":1},
    {"source":62,"target":55,"value":9},
    {"source":62,"target":58,"value":17},
    {"source":62,"target":59,"value":13},
    {"source":62,"target":48,"value":7},
    {"source":62,"target":57,"value":2},
    {"source":62,"target":41,"value":1},
    {"source":62,"target":61,"value":6},
    {"source":62,"target":60,"value":3},
    {"source":63,"target":59,"value":5},
    {"source":63,"target":48,"value":5},
    {"source":63,"target":62,"value":6},
    {"source":63,"target":57,"value":2},
    {"source":63,"target":58,"value":4},
    {"source":63,"target":61,"value":3},
    {"source":63,"target":60,"value":2},
    {"source":63,"target":55,"value":1},
    {"source":64,"target":55,"value":5},
    {"source":64,"target":62,"value":12},
    {"source":64,"target":48,"value":5},
    {"source":64,"target":63,"value":4},
    {"source":64,"target":58,"value":10},
    {"source":64,"target":61,"value":6},
    {"source":64,"target":60,"value":2},
    {"source":64,"target":59,"value":9},
    {"source":64,"target":57,"value":1},
    {"source":64,"target":11,"value":1},
    {"source":65,"target":63,"value":5},
    {"source":65,"target":64,"value":7},
    {"source":65,"target":48,"value":3},
    {"source":65,"target":62,"value":5},
    {"source":65,"target":58,"value":5},
    {"source":65,"target":61,"value":5},
    {"source":65,"target":60,"value":2},
    {"source":65,"target":59,"value":5},
    {"source":65,"target":57,"value":1},
    {"source":65,"target":55,"value":2},
    {"source":66,"target":64,"value":3},
    {"source":66,"target":58,"value":3},
    {"source":66,"target":59,"value":1},
    {"source":66,"target":62,"value":2},
    {"source":66,"target":65,"value":2},
    {"source":66,"target":48,"value":1},
    {"source":66,"target":63,"value":1},
    {"source":66,"target":61,"value":1},
    {"source":66,"target":60,"value":1},
    {"source":67,"target":57,"value":3},
    {"source":68,"target":25,"value":5},
    {"source":68,"target":11,"value":1},
```

```
    {"source":68,"target":24,"value":1},
    {"source":68,"target":27,"value":1},
    {"source":68,"target":48,"value":1},
    {"source":68,"target":41,"value":1},
    {"source":69,"target":25,"value":6},
    {"source":69,"target":68,"value":6},
    {"source":69,"target":11,"value":1},
    {"source":69,"target":24,"value":1},
    {"source":69,"target":27,"value":2},
    {"source":69,"target":48,"value":1},
    {"source":69,"target":41,"value":1},
    {"source":70,"target":25,"value":4},
    {"source":70,"target":69,"value":4},
    {"source":70,"target":68,"value":4},
    {"source":70,"target":11,"value":1},
    {"source":70,"target":24,"value":1},
    {"source":70,"target":27,"value":1},
    {"source":70,"target":41,"value":1},
    {"source":70,"target":58,"value":1},
    {"source":71,"target":27,"value":1},
    {"source":71,"target":69,"value":2},
    {"source":71,"target":68,"value":2},
    {"source":71,"target":70,"value":2},
    {"source":71,"target":11,"value":1},
    {"source":71,"target":48,"value":1},
    {"source":71,"target":41,"value":1},
    {"source":71,"target":25,"value":1},
    {"source":72,"target":26,"value":2},
    {"source":72,"target":27,"value":1},
    {"source":72,"target":11,"value":1},
    {"source":73,"target":48,"value":2},
    {"source":74,"target":48,"value":2},
    {"source":74,"target":73,"value":3},
    {"source":75,"target":69,"value":3},
    {"source":75,"target":68,"value":3},
    {"source":75,"target":25,"value":3},
    {"source":75,"target":48,"value":1},
    {"source":75,"target":41,"value":1},
    {"source":75,"target":70,"value":1},
    {"source":75,"target":71,"value":1},
    {"source":76,"target":64,"value":1},
    {"source":76,"target":65,"value":1},
    {"source":76,"target":66,"value":1},
    {"source":76,"target":63,"value":1},
    {"source":76,"target":62,"value":1},
    {"source":76,"target":48,"value":1},
    {"source":76,"target":58,"value":1}
  ]
}
```

B Collapsible Tree Layout