Logic Circuit Workbench: GatePlay

Greg O'Connor
Computer Science, University of Oxford

Third Year Project
Supervisor: Geraint Jones

May 25, 2014

Abstract

GatePlay is an application to build and simulate logic circuits from within the web browser, without the need for installation or extra plug-ins. It is designed to couple a simple and aesthetically pleasing Graphical User Interface with an efficient and accurate simulation.

Contents

Chapter 1

Introduction

1.1 Motivation

Logic circuits underpin practically every aspect of modern life in the form of computer chips. It is therefore important that people have the opportunity to learn about and understand logic circuits.

I believe that one of the best ways to learn about logic circuits is to build them, and that the most accessible way of doing that is by using a simulated workbench — avoiding the cost and expertise required to build physical circuits. The workbench would not be designed to be used by professionals in designing chips, but for casual users looking to learn about logic circuitry.

Projects which attempt to solve the same problem can already be found on the Internet. Two of the best examples I could find were Logic.ly[1] and CircuitLab[2].

Logic.ly was easy to use but had two major problems. First and foremost, the simulation implemented in Logic.ly over simplified: it does not assign any propagation delay to its components, making it difficult to see what happens in oscillating circuits. Secondly, it is implemented using Adobe Flash. Flash requires a web browser plug-in not included on many devices (notably iOS devices), which reduces Logic.ly's accessibility.

CircuitLab is designed more for power users and has a much steeper learning curve than Logic.ly. CircuitLab allows users to create general electronic circuits from power sources, resistors, etc. which adds to the complexity of the user interface. Even as someone who considers themselves computer literate, I found it difficult to use CircuitLab.

Therefore I decided to build a workbench which is easy to use but would not simplify the simulation more than necessary.

[1]http://logic.ly
[2]http://www.circuitlab.com

1.2 Overview of GatePlay

GatePlay can be seen in figure 1.1. The labelled regions are:

1 The **top bar**, which has buttons to download the workbench as an image and to start simulating the circuit. When simulating, the top bar has additional controls such as starting, restarting, and pausing the simulation.

2 The **left bar**, which has sliding panels containing the components available to build circuits with. The user simply drags a gate from the left bar onto the workbench to add it to the circuit.

3 The **workbench**, which is where almost all interaction with GatePlay happens. When editing you can move, delete, and draw wires between components. When simulating wires are coloured to show their current value.

The components and wires on the workbench snap to the grid lines which can be seen in the screenshots.

GatePlay has a library of standard components which can be used to build circuits. All of the components are implemented as black boxes, even though most of them can be re-implemented using a combination of the simpler components.

- **Input** components such as constant ON, $Toggle$ (clicking on a $Toggle$ during simulation will change its value), and *Blinker* (which flip their value at a constant interval)
- **Output** components such as LEDs
- **Gates** implement standard boolean logic functions such as NOT, AND, and XOR
- **Function** components such half adders and full adders
- **Memory** components such as SR latches and D flip-flop

A typical workflow for creating circuits would be:

1. Drag the desired components from the left-bar on to the workbench
2. Wire the components up by left-clicking on output ports, and then left-clicking on an input port. Fixed points can be created along the way by left-clicking on an empty part of the workbench
3. Entering simulation mode by clicking on the “Start Simulating” button on the top-bar
4. The simulation can now be paused, resumed, or advanced manually by using the controls in the top-bar
5. The user may decide to return to editing mode to add, move, or delete components, or re-wire parts of the circuit

Figure 1.2 shows a D flip-flop being simulated on the workbench. Green wires are $True$ and red wires are $False$. The round components on the left are $Toggle$s; clicking them will change their value and cause the resulting changes to propagate through the circuit. The round components on the right are LEDs, which simply display their input value.

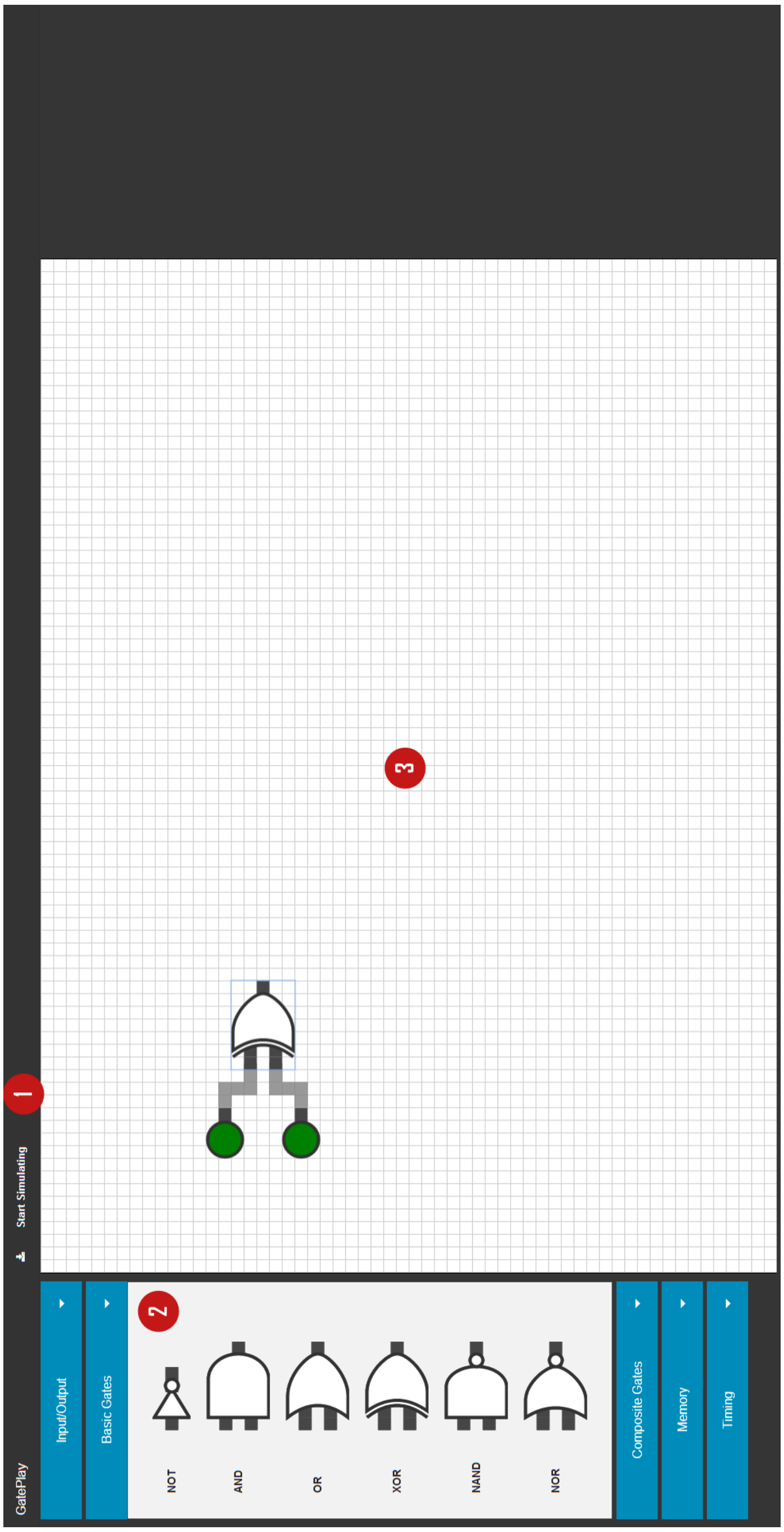

Figure 1.1: Drawing a circuit

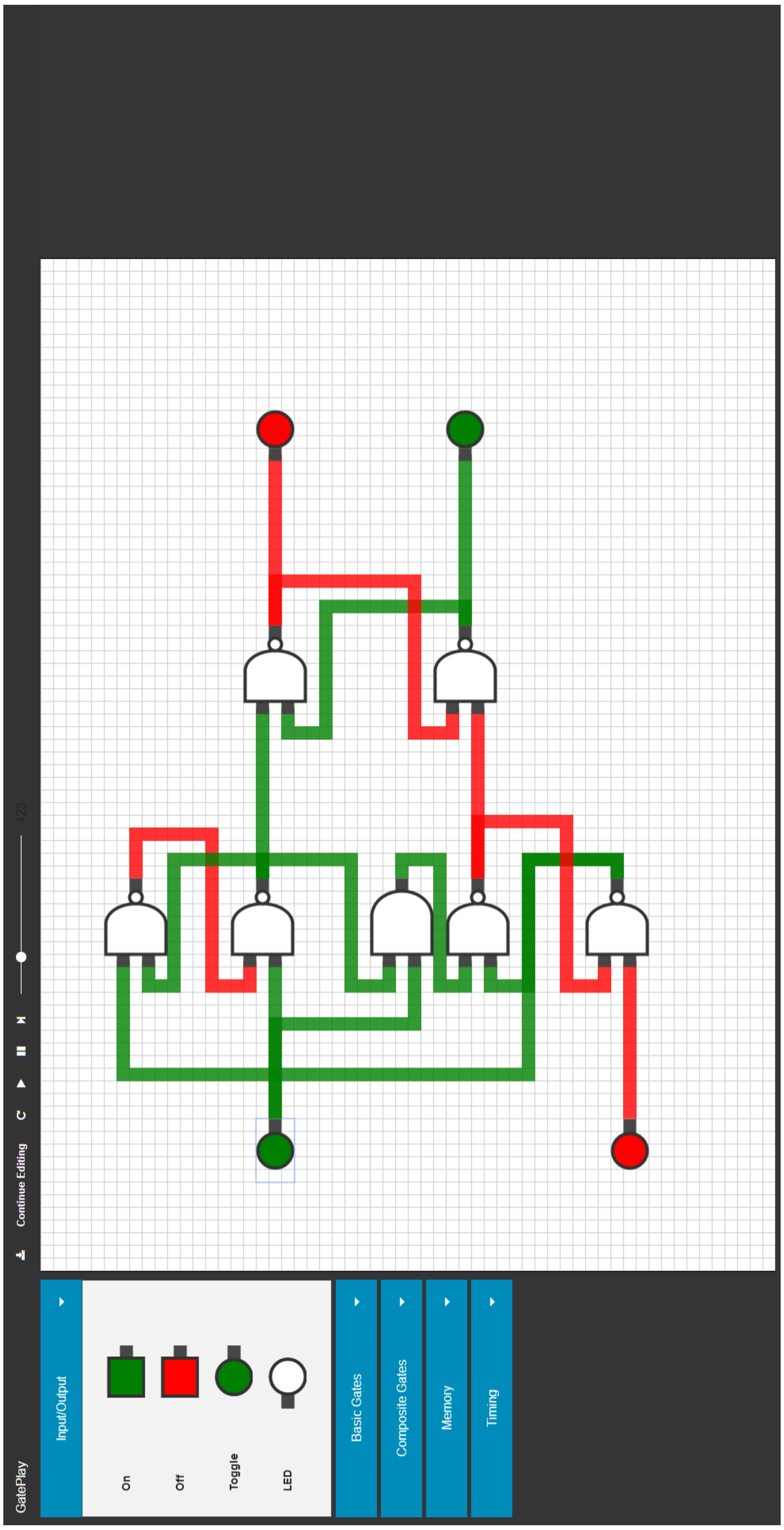

Figure 1.2: Simulating a D flip-flop

Chapter 2

Background

2.1 Logic Circuits

A *k-ary Boolean function* is a function which take k boolean values and returns a boolean value.

$$F : \{True, False\}^k \to \{True, False\}$$

Logic gates are physical implementations of (typically simple) boolean functions. A *logic circuit* is a collection of logic gates wired together, producing an implementation of the composition of boolean functions.

By composing ever more complex functions we can create physical implementations of useful circuits — for example circuits which add or multiply the binary representation of numbers.

A logic gate has a *propagation delay* defined as the time from when its inputs become stable and valid to when its outputs become stable and valid. Propagation delay changes based on temperature, voltage, and output capacitance[1].

2.2 Building Websites

GatePlay is a web application. Instead of being installed on the user's computer it runs in their web browser. GatePlay is a single HTML document and has relatively few CSS style rules. The vast majority of the work was creating the JavaScript, which handles all interactions with the top and left bars, as well as displaying, editing, and simulating circuits.

HTML defines the semantic structure of the website as a collection of nested elements, known as the DOM (Document Object Model) tree. In other words, HTML defines the content of a website and the structure of headings, sections, paragraphs, etc.

CSS Cascading Style Sheets modify how HTML documents are displayed by the client's browser. CSS files are lists of *selectors* with associated *attributes*.

A selector describes elements based on their position in the DOM tree, and its attributes modify how those elements are displayed. For example, it is easy to specify the following styles in CSS: “All elements of type *gate* should be 150 pixels wide”, or “The middle section of the website should take up 80 percent of the width”

JavaScript JavaScript is a full programming language which is run in the user’s browser when they load the website. It can perform arbitrary computations, and is also free to modify the structure of the DOM tree and the styles of elements

2.2.1 JavaScript

JavaScript is as dynamic, imperative programming language which can run in all modern web browsers. The code listings included in this report should also be understandable to those familiar with languages such as C++ or Java.

JavaScript is an example of a Prototype-based programming language[1]. In the code listings of this report, lines such as “MyClass.prototype.myMethod = function()” can be read simply as adding the method *myMethod* to the class definition of *MyClass*.

[1]http://en.wikipedia.org/wiki/Prototype-based_programming

Chapter 3

Requirements

It was agreed with my supervisor that the project should satisfy the following requirements:

- **Accessibility:** The workbench needs to be accessible to as many users as possible. This requirement is broken down into several sub-requirements:
  - **Web based:** The workbench must be implemented in such a way that it is accessible from the Internet, and runs in the user's browser (rather than needing to be downloaded and installed)
  - **Plug-in free:** It should run in the browser without requiring third-party plug-ins such as Adobe Flash or Microsoft Silverlight. This effectively limits the implementation language to JavaScript or a language which can be compiled to JavaScript
  - **Simple UI:** The user interface must be simple to understand and use. Users should be able to use familiar actions such as box selection, deletion using the Delete key, drag-and-drop, etc.
- **Non-simplistic simulation:** While the workbench should be easy to use, it should not simply the simulation beyond what is reasonable. The simulation should take in to account the uncertainty in a gate's propagation delay exhibited by real logic gates

Chapter 4

Simulation

4.1 Modelling Circuits

A circuit can be modelled as a set of components and a set of wires. The fields which define components and wires are shown in figures 4.1 and 4.2 respectively. The only constraint placed on circuits is that no more than one wire may go into the same input port of the same component.

- The **number of inputs** the component has (N)
- The **number of outputs** the component has (M)
- An **evaluation function** which takes a list of N truth values and returns a list of M truth values

Figure 4.1: Fields of a Component

- The **source component** which the wire is leaving from
- The **output port** of the source component
- The **target component** which the wire is going to
- The **input port** of the target component
- The current **truth value** of the wire

Figure 4.2: Fields of a Wire

We say that component X is *wired* to component Y if there exists a wire w in the set of wires such that w's source component is X and w's target component is Y.

We assume for now that each component has a constant propagation delay δ, which is defined by the component's evaluation function. The background section 2.1 states that a gate's propagation delay actually varies, and we will take that into consideration later in this chapter.

4.1.1 2-Valued Simulation

Logic circuits are physical implementations of Boolean functions. The domain of Boolean variables is $\{True, False\}$, so a natural simulation would use the same two truth values for the evaluation functions in our model. In other words, each wire has one of two values: *True* or *False*.

We consider the problems with a 2-valued simulation in section 4.3 and fix them by using a 3-valued simulation in section 4.4, but for now let us set the domain of truth values to be $\{True, False\}$.

4.2 Event-Based Simulation

GatePlay uses an event-based algorithm to simulate logic circuits. An event is a notification that a specific output of a component has changed. They are defined by a 4-tuple (X, P, T, V) where:

- **X** is the **source component** the event is propagating from
- **P** is the **output port** on the source component which has changed value
- **T** is the **event time** at which it is occurring
- **V** is the new **truth value** of the output port

Events are stored in a priority queue, and have priority equal to their event time. Lower times are more urgent.

4.2.1 Initial Components

The first events in a simulation run are generated by what I call *Initial Components*. An initial component is defined as a component which takes no inputs. Some examples of initial components are described below:

- **Constant Components** such as ON and OFF never change value, and so place their events in the queue just once when the circuit is initialised
- **Timed Components** such as *Blinker*s toggle their output at a set interval. The algorithm to implement this is non-trivial and discussed in chapter 5
- **External Components** such as *Toggle*s add events based on user interaction. Since the stimulus to create an event comes from outside the simulator, it is a simple case of putting a method to create circuit events in the circuit's public API

4.2.2 The Event Loop

The heart of an event-based simulation is the event loop which processes all the events and generates new events. The stages of the event loop body are shown in figure 4.3.

1. **Fetch** the next event to be processed from priority queue
2. **Update** the value of the wires connected to the event port
3. **Recalculate** the output of any gates whose inputs changed
4. **Propagate** the change by creating new events for each changed output

Figure 4.3: Event Loop Body

4.2.3 Event Loop Example

Consider the circuit shown in figure 4.4. Let X's current output be $False$, the event queue contain a single event $ev = (X, 1, t, True)$, and the current time be t.

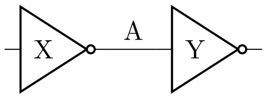

Figure 4.4: Two NOT Gates

When the event loop body executes at simulated time t, the following actions occur:

1. Pop ev from the queue
2. Update A — the wire coming from X port 1 — to be valued $True$
3. A enters Y, so Y's input has changed. Apply Y's evaluation function to the new input, which returns $False$ (as $\neg True = False$)
4. Since Y's output changed, add the following new event to the queue: $(Y, 1, t + \delta, False)$ where δ is Y's propagation delay

As the event loop repeats, changes propagate through a circuit.

4.2.4 Culling

Events can sometimes be discarded without being processed by the entire event loop body. For example, events which set an output port to the value it already is do not change anything in the circuit and can be discarded during stage 2 of the event loop.

4.2.5 Event Race Conditions

Consider the circuit shown in figure 4.5:

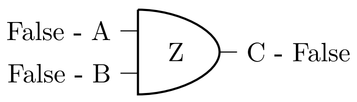

Figure 4.5: AND gate

Let there be two events in the event queue: $ev_1 = (X, 1, t, True)$ and $ev_2 = (Y, 1, t, True)$ where wire A leaves X port 1, and wire B leaves Y port 1. In other words, A and B both become $True$ at simulated time t. Now consider what happens in the event loop at time t.

Suppose we handle ev_1 first: We set A to $True$, B is still thought to be $False$, and therefore we add the event $ev_3 = (Z, 1, t+\delta, False)$ to the queue (as $True \wedge False = False$).

Next we handle ev_2: B is set to $True$ and A is known to be $True$, so we add the event $ev_4 = (Z, 1, t + \delta, True)$ to the queue (as $True \wedge True = True$).

The queue now contains two events with different truth values occurring on Z port 1 at the same time. At simulated time $t + \delta$, if ev_3 is handled first it will be culled and the circuit will be simulated correctly. However if it is handled second then the output of Z will be calculated as $False$, despite both its inputs being $True$! Since we are using a priority queue and both events have the same priority, there is no defined behaviour for which event will be handled first.

The solution is to do a first pass through all events happening at time t and update the values of each of the affected wires. We look at ev_1 at set A to $True$, and at ev_2 and set B to $True$. Now if we process ev_1: A and B are already set to $True$ because of the first pass, and we calculate Z's output to be $True$. Handling ev_2 will again yield $True$, and one of the duplicate events will be culled.

There is a further refinement implemented in GatePlay: when we do the initial pass through all events happening at time t we maintain a set of the affected components. In this example both events only affect Z, so the set will be $\{Z\}$ — sets do not store duplicate values. We then re-calculate the output of each component in the set and add the event to the queue. In this case, we will only calculate and add $(Z, 1, t + \delta, True)$ once.

4.2.6 Efficiency

The speed of an event-based simulation is proportional to the number of events being generated and processed. For this reason, reducing the time spend processing events through Culling (see section 4.2.4) is critically important.

Also note that (if sensible data structures are used) event-based simulations still perform well in circuits with a very large number of components and connections, so long as relatively few events are occurring. An efficient simulation is not necessary for GatePlay as the UI only allows users to build modest-sized circuits, but it is good to implement an efficient simulator regardless in case the UI is ever changed.

4.3 Limitations of 2-Valued Simulations

4.3.1 Initialisation Values

In a 2-valued simulation all wires must be either $True$ or $False$ valued. It is necessary to initialise the wire values before the circuit begins, but what is a sensible default? Suppose we initialise all wires to $False$, and consider the circuit in figure 4.6:

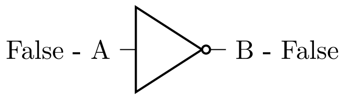

Figure 4.6: An inconsistently initialised *NOT* Gate

To have both A and B both be initialised to be $False$ is inconsistent with the logic of a NOT gate. The same would be true if wires were initialised to $True$. One option would be to add a boolean flag to wires indicating that no event has reach them yet and leave them uninitialised. GatePlay, however, fixes the problem by using a 3-valued simulation (section 4.4).

4.3.2 Propagation Delay Uncertainty

As discussed in section 2.1, the *Propagation Delay* of a component is the time it takes from its inputs being stable and valid to its outputs becoming stable and valid[1].

Previously we have assumed that this delay is constant for a given component, but in reality the precise delay varies based on temperature, voltage, and output capacitance[1]. Our model of logic circuits does not consider these factors and therefore cannot make an informed estimation of the true delay for each pass through a component.

We assume that the propagation delay is never less than δ_{min} and never greater than δ_{max}. δ_{min} and δ_{max} can be different for different types of components.

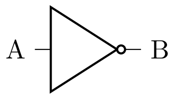

Figure 4.7: A *NOT* Gate

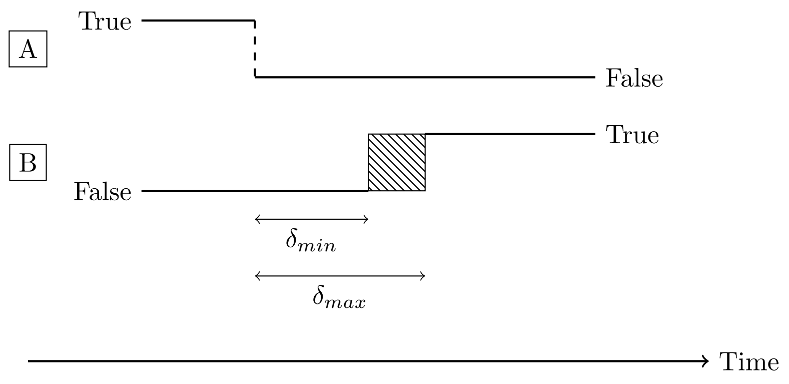

Figure 4.8: Trace of a *NOT* gate's input and output

Consider the circuit shown in figure 4.7. If we flip A's value from *True* to *False* the trace of the truth values through time is shown in figure 4.8. *B*'s transition from *False* to *True* can happen at any point in hatched region based on the aforementioned factors. A 2-valued simulator at this level of detail can therefore only assume that the transition happens at a random time. In other words, the precise propagation delay for each pass through a component is randomly sampled from the interval $[\delta_{min}, \delta_{max}]$.

Each simulation of a circuit will likely have different propagation delays which can potentially change the output of a circuit altogether. If one run of this simulation yields a given result there is no guarantee that *all* runs would yield the same result. I felt this was not desirable behaviour for a simulator. In the next section we introduce a simulator which models the uncertainly of components without introducing non-determinism in the simulation itself.

4.4 3-Valued Simulation

The way I decided to overcome the problems of 2-valued simulation described in section 4.3 was to instead use a 3-valued simulation. A wire has one of three values: *True*, *False*, or *Unknown*. A wire that is *Unknown* is "in reality" either *True* or *False*, but the simulation does not know which.

4.4.1 Initialisation Values

In a 3-valued simulation all wires can be initialised to *Unknown* without the problem of inconsistency for 2-valued simulations discussed in section 4.3.1.

4.4.2 Propagation Delay Uncertainty

The 2-valued simulation was unable to model the uncertainty of component delay without making the result of each simulation run non-deterministic, as discussed in section 4.3.2.

Suppose we have a component X whose inputs are changing at time t. In the 2-valued simulation we would consider the outputs changing at time $t+\delta$ where δ is randomly sampled from the interval $[\delta_{min}, \delta_{max}]$.

However in the 3-valued simulation we can consider the outputs of X to be changing twice. At time δ_{min} they becoming *Unknown*, and then at time δ_{max} they settle on a stable value.

By considering the output *Unknown* for the duration of the delay uncertainty we have still modelled the uncertainty in the actual components, but without introducing uncertainty to our simulation. In other words, should a circuit yield a certain value in our simulation we know that it will yield the same value for *all* possible combinations of gate delays.

Evaluation functions must also be augmented to accept *Unknown* as in input, and potentially produce *Unknown* as an output. For example, we define $True \wedge Unknown = Unknown$, $Unknown \wedge Unknown = Unknown$, and $False \wedge Unknown = False$.

4.4.3 Event Suppression

A bug was introduced in the simulator in section 4.4.2. Consider the circuit shown in figure 4.9. Let δ_{min} and δ_{max} be the minimum and maximum gate delays respectively. A, B, and C are all $False$.

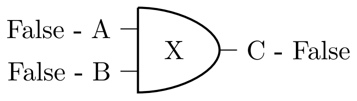

Figure 4.9: An *AND* Gate

Let circuit event e_1 set A to $True$ at time t, and event e_2 set B to $True$ at time $t+1$. It is now simulated time t. Using the algorithm thus far described we will handle e_1 and the following two events will be added to the queue: $e_3 = (X, 1, t + \delta_{min}, Unknown)$, $e_4 = (X, 1, t + \delta_{max}, False)$. Next, handling e_2 we will add the following events to the queue: $e_5 = (X, 1, t{+}1{+}\delta_{min}, Unknown)$, $e_6 = (X, 1, t + 1 + \delta_{max}, True)$.

The bug arises at simulated time $t+\delta_{max}$ when we handle e_4. At that time C is set to *Unknown* and we will set it to $False$. However this is incorrect behaviour as e_2 means that C should be uncertain until e_6 is handled at time $t+1+\delta_{max}$.

In other words, the output of X is uncertain for two distinct reasons. e_1 causes C to be uncertain in the interval $[t + \delta_{min}, t + \delta_{max})$, and e_2 makes C uncertain in the interval $[t + 1 + \delta_{min}, t + 1 + \delta_{max})$.

The simulator therefore must keep track of the intervals a wire is uncertain for, and *suppress* any events which try to set it to a certain value in any of those intervals. In this example e_4 must be suppressed.

Chapter 5

Implementation

GatePlay is implemented in the HTML/CSS/JavaScript stack. A brief overview of what HTML, CSS, and JavaScript are is provided in section 2. This chapter will go through the main JavaScript files in the project and the relationships between them.

Some of the JavaScript files are not my own work, but open-source libraries. Re-using others' code lets me focus on implementing the parts unique to GatePlay and avoid writing boilerplate code[1]. The most important libraries are described throughout this chapter.

The implementation is split in to two main, independent parts: the workbench (described in section 5.2) and the simulator (described in section 5.3). I explain how I linked the parts together in section 5.5.

5.1 Getting Started

GatePlay is made up of many JavaScript files, but when its URL is visited the only file downloaded is the main HTML file: *index.html*. The index must then direct the user's browser to download the additional files needed to run GatePlay.

JavaScript files are loaded into the browser using HTML Script tags, in the order the Script tags appear in the document. JavaScript scripts may use methods or variables defined in other scripts provided that those scripts have already been loaded and run in advance.

That means that if *model.js* and *view.js* are JavaScript scripts, and *view.js* depends on *model.js*, then *model.js*'s Script tag must appear first or the website will fail to load. An example is shown in figure 5.1.

[1]http://en.wikipedia.org/wiki/Boilerplate_code

```html
<!-- componentview depends on component -->
<!-- Therefore we ensure component is loaded first -->
<script type="text/javascript" src=".../component.js"></script>
<script type="text/javascript" src=".../componentview.js"></script>
```

Figure 5.1: Example use of Script tags

5.1.1 Require.js

It is time consuming for a human to find and type out a correct ordering for the Script tags, and it would need to be updated every time a file is added or removed, or sometimes if a file were modified.

Require.js is a JavaScript file loader which automatically loads files in a correct order. Each JavaScript file declares its direct dependencies, and Require.js will ensure they are all loaded correctly when the website loads.

Every JavaScript file it GatePlay is configured to work with Require.js, but code listings after this point will omit the Require.js machinery for the sake of clarity.

```javascript
// componentview.js

require([
  // Declare the path of each file we require
  "canvas/model/component"
], function(Component) {
  // Each included file is run, and we can give a name to whatever
      it returns if desired
  var myComponent = new Component();
  ...
});
```

Figure 5.2: An example file which uses Require.js

5.2 The Workbench

GatePlay's workbench is where we create and view circuits. I used the model-view-controller[2] design pattern to simply development. MVC programs are split in to three types of component:

- **Models** which store some part of the state of the application
- **Views** which display a representation of one of the models to the user
- **Controllers** which process user input and updates the appropriate models

The interactions between the components can be seen in figure 5.3. By separating out the concerns of the program it becomes possible to, for example, add new views for your models without needing to change the models or controllers themselves.

[2]http://en.wikipedia.org/wiki/Model-view-controller

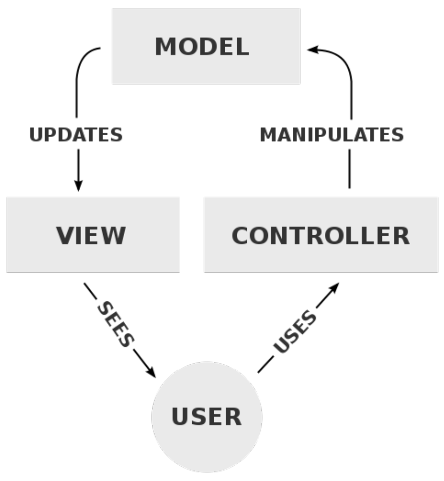

Figure 5.3: Interaction of MVC Components, from Wikipedia[2]

5.2.1 MVC with Backbone.js

Backbone.js[3] is a JavaScript library used to reduce the amount of boilerplate code in developing MVC JavaScript applications by providing predefined notions of models, views, and controllers. A specific example is that Backbone models automatically fire events when any of their fields change value, so I did not have to reimplement that myself.

5.2.2 Models

I first created the models behind the workbench. The top-level model is a *Circuit* model which contains a set of *Component* models and a set of *Wire* models.

Part of the Component model definition is shown in figure 5.4. The actual model contains some additional helper methods, but they do not add any new state to the model and are not shown. Each Component is given a unique identifier (id), an (x, y) position on the workbench, a width, and input and output counts. The *isValid* flag is set to False if this component is ever over another component (which the Circuit disallows) so it can be drawn in a different colour. *activeInputIndex* and *activeOutputIndex* are set when the user hovers their mouse over an input or an output port, so they may be highlighted in a different colour.

Notice that the height of a Component is a function of its input and output counts.

Next I created the Wire model (figure 5.5). The Wire model is very close to the definition of a wire we used in the simulator, except we additionally store a list of *fixed points*. Fixed points are points through which the wire passes on its

[3]http://backbonejs.org

```
var Component = Backbone.Model.extend({
  defaults: function() {
      return {
          id: nextId++,
            x: 0,
            y: 0,
            width: 7,
            inputCount: 2,
            outputCount: 1,
            isValid: true,
            activeInputIndex: -1,
            activeOutputIndex: -1
        }
    },

    getHeight: function() {
        return Math.max(this.get("inputCount"), this.get("outputCount")) * 2 + 1;
    }
});
```

Figure 5.4: Definition of a Component model

```
var Wire = Backbone.Model.extend({
  defaults: function() {
      return {
          id: nextWireId++,
          sourceId: -1,
          sourcePort: -1,
          targetId: -1,
          targetPort: -1,
          fixedPoints: [],
          truthValue: TruthValue.UNKNOWN
      }
  },
})
```

Figure 5.5: Definition of a Wire model

way from the source component to the target component. Wire segments must be aligned with the x- or y-axis, and the model checks this condition when it is initialised or whenever the fixed points change.

With the Component and Wire models we can create a Circuit model. I will not include the Circuit model code listing as it is long yet simple, but rather describe how it works below.

As in the simulator, a Circuit on the workbench contains a set of Components and a Set of Wires. The workbench Circuit also has a given width and height which all components and wires must be contained within.

In GatePlay components are not permitted to overlap each other, and it is the Circuit model's responsibility to enforce this. Recall that the Circuit has a given width and height, and each component has a position, width, and height. The Circuit maintains a 2D array where each cell is a point on the workbench, containing the identifier of the component at that position (or an empty flag if no component). When adding a component to the circuit, it calculates the points the new component will occupy and then checks that no component already occupies those cells. If the check fails, then the component will not be added to the model.

5.2.3 Views

Now that we have the back end models for the workbench, it is a matter of displaying them to the user.

To display circuits we need an element on the web page to draw graphics. I used an HTML Canvas element — a blank slate on which we can draw shapes and images — as it provides the exact functionality required.

The CircuitView will initialise the canvas, and then delegate to a collection of ComponentViews and WireViews to draw the Components and Wires respectively.

Fabric.js

From the user's perspective the workbench is going to contain a collection of objects (components), each of which can be selected and moved. An HTML Canvas is essentially a flat array of pixels and does have any concept of objects on the canvas.

Fabric.js is an canvas framework which wraps HTML Canvas elements with an object model, allowing GatePlay to interact at the level of objects being added to, modified on, and removed from the canvas.

As an example, consider the user moving a component from one position on the workbench to another. With an HTML Canvas I would need to re-calculate and re-draw the background over the component, and then re-draw the component at its new location. Fabric.js already implements this feature, and it is a simple library call to move the component to a new location. Using Fabric.js greatly reduces the amount of code needed to interact with canvases.

Initially GatePlay used a different canvas framework called KineticJS, but due to difficulties getting features like snap-to-grid working I switched to Fabric.js early on in development.

ComponentView

An instance of the ComponentView class is responsible for drawing a single instance of a Component model. On instantiation, a ComponentView is passed the canvas it is to draw on and the model it is to draw. The code of the ComponentView is long and simple. Figure 5.6 contains a reduced version of the code, and I will explain the meaning of each method below. The *this.model.on* calls in the *initialize* method are examples of Backbone.js reducing boilerplate code: when the model is changed the view will update automatically.

```
Backbone.View.extend({
    initialize: function(options) {
        this.options = options.options;
        this.model = options.model;
        this._template = null;
        this._activePort = null;

    // When the model changes, update the view
        this.model.on("change:isValid", this._setFillColor, this);
        this.model.on("change:activeInputIndex", this._activeInputChanged, this);
        this.model.on("change:activeOutputIndex", this._activeOutputChanged, this);
        this.model.on("change:truthValue", this._setFillColor, this);
    },

    render : function() {
        // Render the gate on the canvas
    },

    _setFillColor: function() {
        // Set the fill colour of the component
        // LED's are filled the the colour of their input, for example
    },

    _activeInputChanged: function() {
        // The user is hovering over an input port, so colour it red
    },

    _activeOutputChanged: function() {
        // The user is hovering over an output port, so colour it red
    },
});
```

Figure 5.6: Definition of a ComponentView

- The **render** method draws the component on the canvas. It requests a template for the component's type from a repository of templates (there is a template for *AND* gates, and *NOT* gates, and so on) and adds it to the canvas. It also draws the nodes for the input and output ports
- The **setFillColor** method is called when the model's truth value changes. For example *Toggle*s are coloured with the value of their current output, and this method is called when their value changes
- The **activeInputChanged** and **activeOutputChanged** methods are called when the user hovers over an input or output port respectively. It highlights the hovered port with a different colour so users know that clicking on it will perform an action (starting to draw wires)

```
render: function() {
  // Clear old canvas
    this.options.canvas.clear();

    // Draw the components on the grid
    var componentSetView = new ComponentSetView(this.options);
    componentSetView.render();

    // Draw the wires on the grid
    var wireSetView = new WireSetView(this.options);
    wireSetView.render();
},
```

Figure 5.7: CircuitView render method

WireView

An instance of the WireView class draws a single Wire model on the canvas. It is very similar to the ComponentView class in that it has a *render* method and a *setWireColor* method (which is called when the wire value changes). The render function draws each segment between fixed points as a straight line and combines them to create one Fabric.js object which is added to the canvas.

CircuitView

The CircuitView draws the entire circuit. Its *render* method is shown in figure 5.7. When the Circuit is rendered, it simply clears the old canvas, and then redraws the components and wires. The SetViews seen in the code listing simply take a set of models and create a view for each model in turn.

5.2.4 Controllers

There is one controller for GatePlay's workbench, simply called *CircuitController*. It handles all the mouse events which occur on the workbench: mouse movement, mouse clicks, hovering over components, and creating box selections.

In fact it delegates all the handling to its event handler. When the user is editing a circuit it is an instance of EditingEventHandler, when running a simulation it is an instance of RunningEventHandler.

EditingEventHandler

The EditingEventHandler is by far the biggest class in GatePlay and so does not have a code listing. I will briefly describe how it performs each of its responsibilities:

- **Moving components:** If the user's mouse was clicked over an object (not on an output port), held down, and moved then that object is being dragged. The handler updates the position of the component model in real time

- **Drawing wires:** If the user's mouse was clicked over one of an object's output ports, then the user is drawing a wire. Single clicks create a fixed point for the wire, and double clicking cancels drawing. If the user clicks on an empty input port, the appropriate wire is created and added to the model
- **Components selected:** GatePlay allows users to drag components over other components temporarily (colouring them red to show it is an invalid position). The component being moved should always appear on top. To implement this, when components are selected they are immediately brought to the front of the canvas

RunningEventHandler

When running a simulation should not be possible to move components or draw wires. The only event the RunningEventHandler looks for is the user clicking on a *Toggle* component. When this happens, the Component model for that *Toggle* is updated with the new truth value and the simulator is notified of the change.

5.3 The Simulator

The implementation of GatePlay's simulator is a relatively straightforward implementation of the algorithms and ideas explained in chapter 4. The following six classes fully implement the simulator:

- **truthvalue.js** defines constants *True*, *False*, and *Unknown*
- **component.js** defines a Component by its input count, output count, and evaluation function
- **wire.js** defines a Wire by its input component and port, output component and port, and truth value
- **circuitevent.js** defines a CircuitEvent by component, port, timestamp, and value
- **functions.js** contains definitions of all the Evaluation Functions available to the simulator
- **circuit.js** is the only class which need be visible from outside the simulator. It has an interface to add components and wires. Circuit.js implements the algorithm for the event loop.

5.3.1 Blinker Events

Recall section 4.2.1 that *Blinker* components toggle their output value at a set interval. Each *Blinker* has its own, potentially unique, interval.

```
Blinker.prototype._doEvaluate = function(argList, clock) {
    var era = Math.floor(clock / this._interval);
    var parity = era % 2;
    if (parity === 0) {
        return [TruthValue.TRUE];
    } else {
        return [TruthValue.FALSE];
    }
};
```

Figure 5.8: Implementation of *Blinker*'s evaluation function

```
function tick() {
  // For each blinker event which is happening at this time
  while (this._blinkerEventQueue.peek().time <= this._clock) {
    var event = this._blinkerEventQueue.pop();
    this._addEvent(event);

    var blinker = this.getComponent(event.sourceId);
    var nextTime = event.eventTime + blinker.get("interval");
    var nextValue = blinker.evaluate([], nextTime);
    var nextEvent = new CircuitEvent(nextTime, event.sourceId,
        event.sourcePort, nextValue);
    this._blinkerEventQueue.add(nextEvent);
  }

  // Event loop goes here
}
```

Figure 5.9:

To implement this, a *Blinker* component's evaluation function (shown in figure 5.8 determines it truth value based on the simulation clock. Since *Blinker*s have no inputs they will never be processed by the event loop, so we need to handle *Blinker*s differently.

The solution implemented in GatePlay is to add a new circuit event for every *Blinker*, *every* clock tick. For a *Blinker* with interval 2, we would add *True* events at simulated time 0 and 1, *False* events at simulated time 2 and 3, and so on. We rely on culling to eliminate the unnecessary events.

The algorithm described above if very easy to implement, but clearly inefficient if there were a large number of *Blinker*s. A better algorithm where we only add one circuit event per *Blinker* per interval is outlined in figure 5.9. We store in a priority queue the next event for each *Blinker*. Whenever we pop an event from the queue we replace it with an event for the next time the corresponding blinker will change value.

Note that the *blinkerEventQueue* is just a subset of the main event queue, and we could actually merge this code with the event loop. However doing so would couple our implementation of the general event queue with that of a specific type of component and not be good software engineering practice.

5.4 Drag-and-drop

One of the requirements of GatePlay was that it be easy to use, and being able to drag-and-drop components from the left-bar is important in satisfying that requirement. Implementing drag-and-drop is can be split in to three main sub problems:

1. Drawing the components in the left-bar
2. Allow dragging components from the left side-bar
3. Adding components to the workbench where they dropped

To draw the components on the left-bar I create an HTML Image element for each type of component. When the page loads, GatePlay creates an image of each type of Component and displays the image in one of the Image elements. Creating images for each component is handled by the *createThumbnail* function: *createThumbnail* creates a temporary canvas for each component and renders the component on it. This canvas can be converted to an image file by a Fabric.js library call.

Dragging and dropping is an interaction supported by the jQueryUI[4] library, which GatePlay uses. The components in the left-bar are marked as *Draggable* and the main workbench is marked as *Droppable*. With a small amount of configuration, components can be dragged from the left-bar on to the workbench.

Adding the components to the workbench is done by handling jQueryUI's *drop* event which is called when components are dropped on the workbench. The handler converts the location from global co-ordinates (number of pixels from the top-left hand corner of the browser window) to workbench co-ordinates (number of grid cells from the top-left hand corner of the workbench), and instructs the workbench to add a new component there. This is done by the Application class described in the next section.

5.5 Tying GatePlay Together

The implementation discussed so far has multiple distinct parts which have no knowledge of each other: the workbench, the simulator, and droppable interactions. We create a new class *Application* which ties the pieces together as shown in figure 5.10

[4]http://http://jqueryui.com/

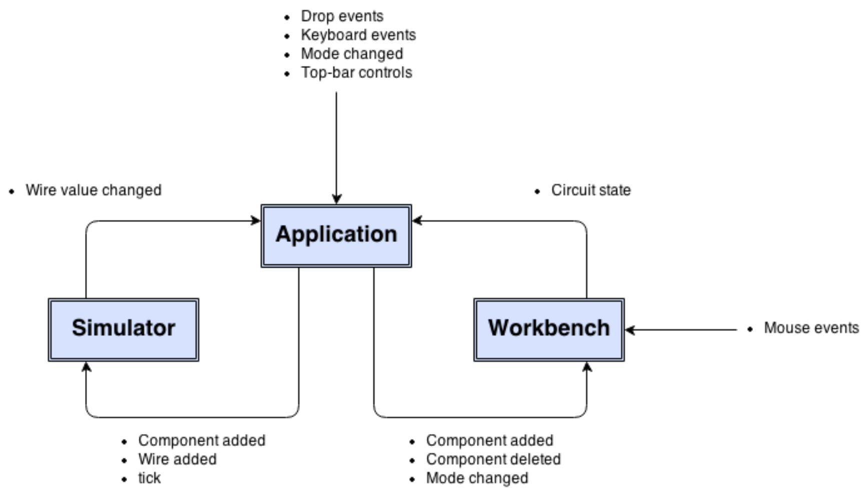

Figure 5.10: Information flow in GatePlay

The Application is the first part of GatePlay's code to be executed, and creates the workbench Circuit model and CircuitView view when it is initialised. It is Application which handles jQueryUI's *drop* event, listens for presses of the Delete key, and notifies the workbench when the mode change to Editing or Running.

When the user clicks to start simulating their circuit, Application copies across all the circuit information from the workbench into a new simulator object (the code listing is shown in figure 5.11). It also registers an event handler with the simulator to update the value of the wires on the workbench when they change in the simulator.

By making GatePlay modular in this way, it is very easy to change one part of the code without changing the rest of it.

```javascript
ApplicationState.prototype._nowRunning = function() {
    this._setClock(0);

    // Create new simulator
    var simulation = new SimCircuit();
    this._simulation = simulation;

    // Add components
    var components = this._canvasModel.get("components");
    _.each(components.models, function(c) {
        simulation.addComponent(c.get("id"), c.get("templateId"), c
            .get("inputCount"), c.get("outputCount"), c.get("cArg")
            );

        if (c.get("templateId") === "toggle" || c.get("templateId")
             === "blinker") {
            c.set("truthValue", "True");
        } else if (c.get("templateId") === "led") {
            c.set("truthValue", "Unknown");
        }
    })

    // Add wires
    var wires = this._canvasModel.get("wires");
    _.each(wires.models, function(wire) {
        simulation.addWire(wire.get("id"), wire.get("sourceId"),
            wire.get("sourcePort"), wire.get("targetId"), wire.get(
            "targetPort"));

        wire.set("truthValue", "Unknown");
    })

    simulation.initialize();
    simulation.addWireEventListener(function(id, truthValue) {
        var wire = wires.get(id);
        wire.set("truthValue", truthValue);

        var source = components.get(wire.get("sourceId"));
        if (source.get("templateId") === "toggle" || source.get("
            templateId") === "blinker") {
            source.set("truthValue", truthValue);
        }

        var target = components.get(wire.get("targetId"));
        if (target.get("templateId") === "led") {
            target.set("truthValue", truthValue);
        }
    })
};
```

Figure 5.11: Application.js's nowRunning method

Chapter 6

Testing

Testing is extremely important in the development of non-trivial programs. Tests give some assurance to the correctness of the code, and highlight *regressions* (where code that used to work is broken by a recent change) quickly.

6.1 Unit Testing

Unit testing is used to test the correctness of small modules of code, such as functions. The simulator has a suite of unit tests. One example is a test over the *AND* evaluation function shown in figure 6.1. All *evaluate* functions had to be changed when *Blinker*s were implemented, and this test flagged a regression when the first implementation had a bug.

Unit tests can also be used to verify more complicated units of code. Figure 6.2 shows a unit test which verifies the output of a simulation run on a full circuit (two *ON*s, an *OR*, and an *LED*).

I used a common JavaScript unit testing framework called QUnit[1] to reduce the boilerplate in writing unit tests.

6.2 End-to-end Testing

Unit testing is a bottom-up approach which checks that the smallest modules work as expected in isolation. End-to-end testing is a top-down approach which tests that entire application works as desired.

Frameworks such as Nightwatch.js[2] do exist to automate end-to-end testing. I felt that writing a comprehensive suite of end-to-end tests would take prohibitively long (there are a huge number of different canvas interactions). Therefore the end-to-end tests were done manually by me.

[1]http://qunit.com

[2]http://http://nightwatchjs.org/

```
var T = TruthValue.TRUE;
var F = TruthValue.FALSE;
var U = TruthValue.UNKNOWN;

function ANDTest() {
  var and = Functions.get("and");

  // Test 2-input truth table
  equal(and.evaluate([T,T]), T);
  equal(and.evaluate([T,F]), F);
  equal(and.evaluate([T,U]), U);
  equal(and.evaluate([F,T]), F);
  equal(and.evaluate([F,F]), F);
  equal(and.evaluate([F,U]), F);
  equal(and.evaluate([U,T]), U);
  equal(and.evaluate([U,F]), F);
  equal(and.evaluate([U,U]), U);
}
```

Figure 6.1: Testing *AND*'s truth table

6.2.1 Ring Oscillator

A ring oscillator[3] can be implemented as an odd number of *NOT* gates arranged in a loop, as shown in figure 6.3.

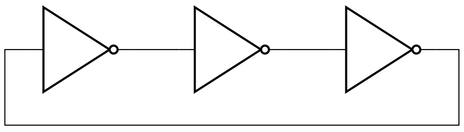

Figure 6.3: A ring oscillator

As discussed in section 4.3.2, the delay of components is not constant. Therefore the period of the oscillator — the sum of the gate delays of the components — is also not constant. After some number of oscillators it is impossible to know the value of any of the wires given only the initial state. If the components were running fast ($\delta = \delta_{min}$) a wire will be one value, if the components were running slow ($\delta = \delta_{max}$) it will be the other.

An explicit requirement of workbench was that it simulate the oscillator falling into the state where all the wire values are *Unknown*. The most common end-to-end test I performed was to build a ring oscillator on the workbench and simulate it. The simulator implemented in GatePlay satisfies this requirement.

6.3 Fail-Fast Methodology

Fail-fast is not a testing strategy, but I mention it here because it helps in reducing the number of undiscovered bugs in a program.

[3]http://en.wikipedia.org/wiki/ring_oscillator

```javascript
function simpleCircuitTest() {
  var circuit = new SimCircuit();

  // Create a simple circuit with 2 ONs, an OR, and an LED
  // Arguments are ID, evalFuncID, inputCount, outputCount
  circuit.addComponent(1, "on", 0, 1);
  circuit.addComponent(2, "on", 0, 1);
  circuit.addComponent(3, "or", 2, 1);
  circuit.addComponent(4, "led", 1, 0);

  // Add wires
  // Id, sourceId, sourcePort, targetId, targetPort
  circuit.addWire(1, 1, 0, 3, 0);
  circuit.addWire(2, 2, 0, 3, 1);
  circuit.addWire(3, 3, 0, 4, 0);

  var wireEvents = [];
  circuit.addWireEventListener(function(wireId, truthValue) {
    wireEvents.push({id: wireId, value: truthValue});
  });

  circuit.initialize();
  for (var i = 0; i < 1000; i++) {
    circuit.tick();
  }

  // Wires should have changed value 3 times in total
  equal(wireEvents.length, 3);

  var event0 = wireEvents[0];
  var event1 = wireEvents[1];
  var event2 = wireEvents[2];

  // The first two events could be in either order and still be
  //correct, so swap them if necessary
  if (event0.id !== 1) {
    var t = event0;
    event0 = event1;
    event1 = t;
  }

  // The first two events are wires 1 and 2 changing U -> T
  equal(event0.id, 1);
  equal(event0.value, TruthValue.TRUE);
  equal(event1.id, 2);
  equal(event1.value, TruthValue.TRUE);

  // The third event must be wire 3 becoming true
  equal(event2.id, 3);
  equal(event2.value, TruthValue.TRUE);
}
```

Figure 6.2: Testing a simple circuit

Consider a method *getComponentById* in a the workbench *Circuit* class. Given some ID of a component, it returns the full object. There are two reasonable behaviours should no such component exist:

1. Return a default value, such as *null*
2. Throw an exception

Suppose *null* is returned and the result is saved to a variable without being checked. It is possible for the program to fail much later in the future when the variable is accessed. The failure may be difficult to debug in the future, because the incorrect code ran much earlier.

Throwing an exception forces the calling code to handle the failure immediately, or the program will halt immediately. By ensuring that errors surface as quickly as possible, bugs can be found sooner and are easier to diagnose.

Chapter 7

Conclusions

Evaluating GatePlay against each of the requirements stated in chapter 3 I reach the following conclusions:

Accessibility GatePlay runs in all modern web browsers without the need for plug-ins, and satisfies those aspects of the requirements. The feedback I received from those who saw GatePlay running was that the user interface was appealing and simple to use. Some people found the workflow for creating wires slightly unintuitive (you single click to start drawing, rather than click and hold), so a miniature tutorial to guide new users through the interface would certainly be of benefit.

Simulation I am pleased with the level of simulation implemented in GatePlay. Having gate propagation delays allows users to see clearly how changes in the inputs to a circuit propagate through to the outputs. I am also satisfied that users of GatePlay will not come away with an over-simplified view of logic circuits which might harm their understanding in the future.

7.0.1 Future Work

While I am very pleased with GatePlay as it is today, there are some features that would improve it further.

Wires over components

It was always my intention that wires should be able to cross each other, but not cross over components. The workbench Circuit model is able to check if wires cross over any components, but the handler code for moving components became extremely complicated when it had to account for the position of wires. Ultimately I did not resolve the issue, and wires can travel over components.

Moving wires when components are moved

When a component is moved, all the wires to and from that component are deleted. I considered two other strategies for preserving the wires: either creating fixed points where the component ports used to be, or using A* search to find a new path for each wire. In the end I implemented neither. In the future I would like to implement a better strategy for moving wires when a component moves.

Saving circuits as black-boxes

I useful feature would be the ability to create circuits and then save them for use as a black-box later. GatePlay has an existing notion of inputs (*Toggle*s) and outputs (*LED*s) and it would not be much more work to allow users to "compile" their workbench to a black-box.

Simulation oscilloscope

The circuit simulator emits a stream of wire value changes as it runs. In GatePlay, these are handled by the Application class and used to colour the wires on the workbench. In addition to this, it would be useful to be able to view traces of the wire values through time, and I would have liked to write a module to create such traces.

Bibliography

[1] http://en.wikipedia.org/wiki/Propagation_delay

[2] http://en.wikipedia.org/wiki/File:MVC-Process.svg