Logic Circuit Workbench: GatePlay

Greg O'Connor
Computer Science, University of Oxford

May 22, 2014

Abstract

GatePlay is a web application to build and simulate logic circuits from within the web browser, without the need for installation or proprietary 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. Therefore it is 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. The workbench should be very simple to use, but not 'dumb down' the simulation to the point of being unhelpful.

On-line logic circuit workbenches already exist. Two of the best examples I could find on the Internet were Logic.ly[1] and CircuitLab[2].

Logic.ly was easy to use, but had two major problems. Firstly it was implemented using Adobe Flash, which requires a web browser plugin not included on many devices (notably iOS devices). Flash also presents a security risk as historically older versions of the plugin have proved easy to exploit. Secondly, Logic.ly uses a simplistic simulation which does not take into account gate delays.

CircuitLab

1.2 Overview of GatePlay

The main interface of 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.

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

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 the values of wires are shown visually on the workbench.

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)
- **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 $High$ (logical $True$) and red wires are Low (logical $False$). 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.

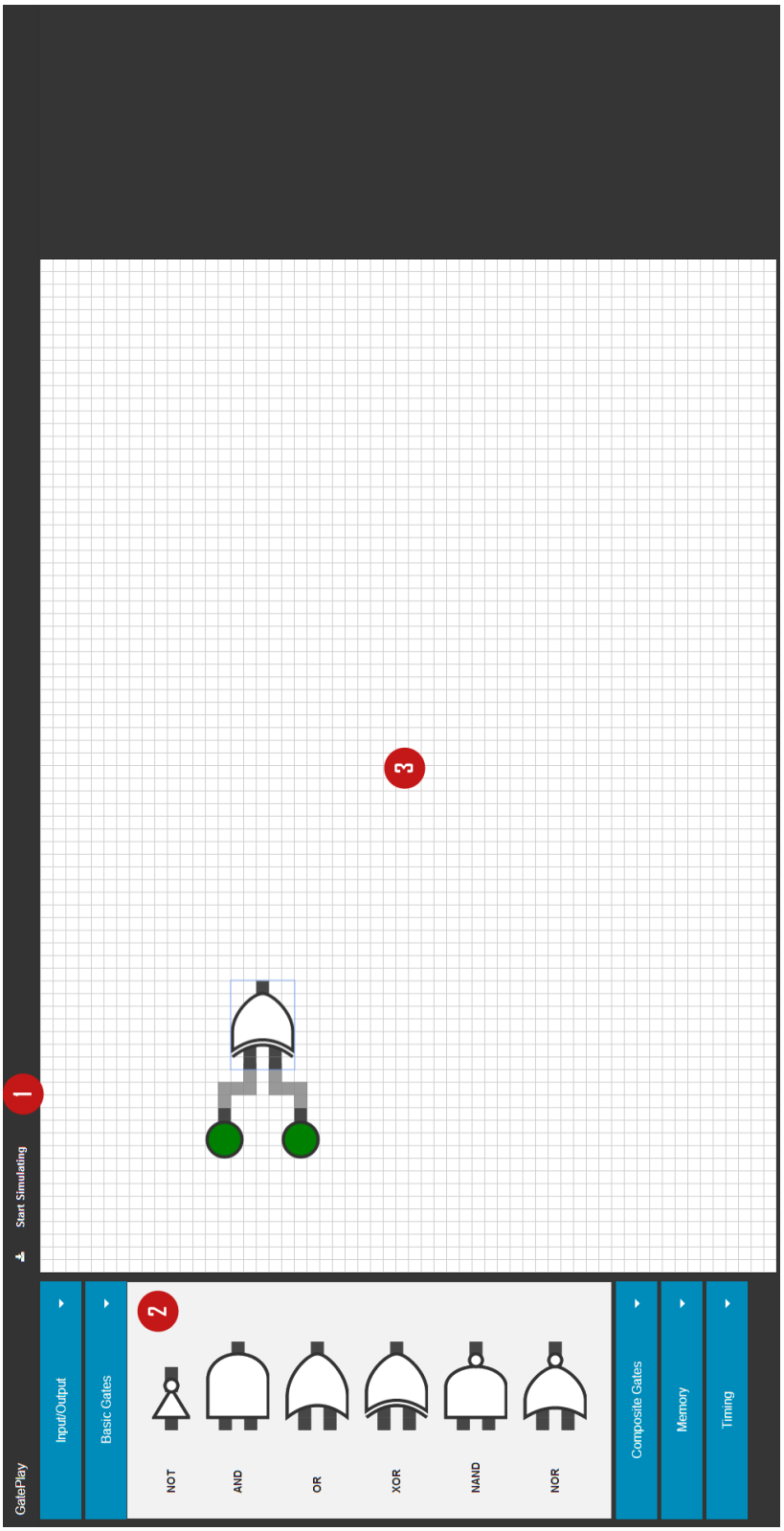

Figure 1.1: Drawing a circuit

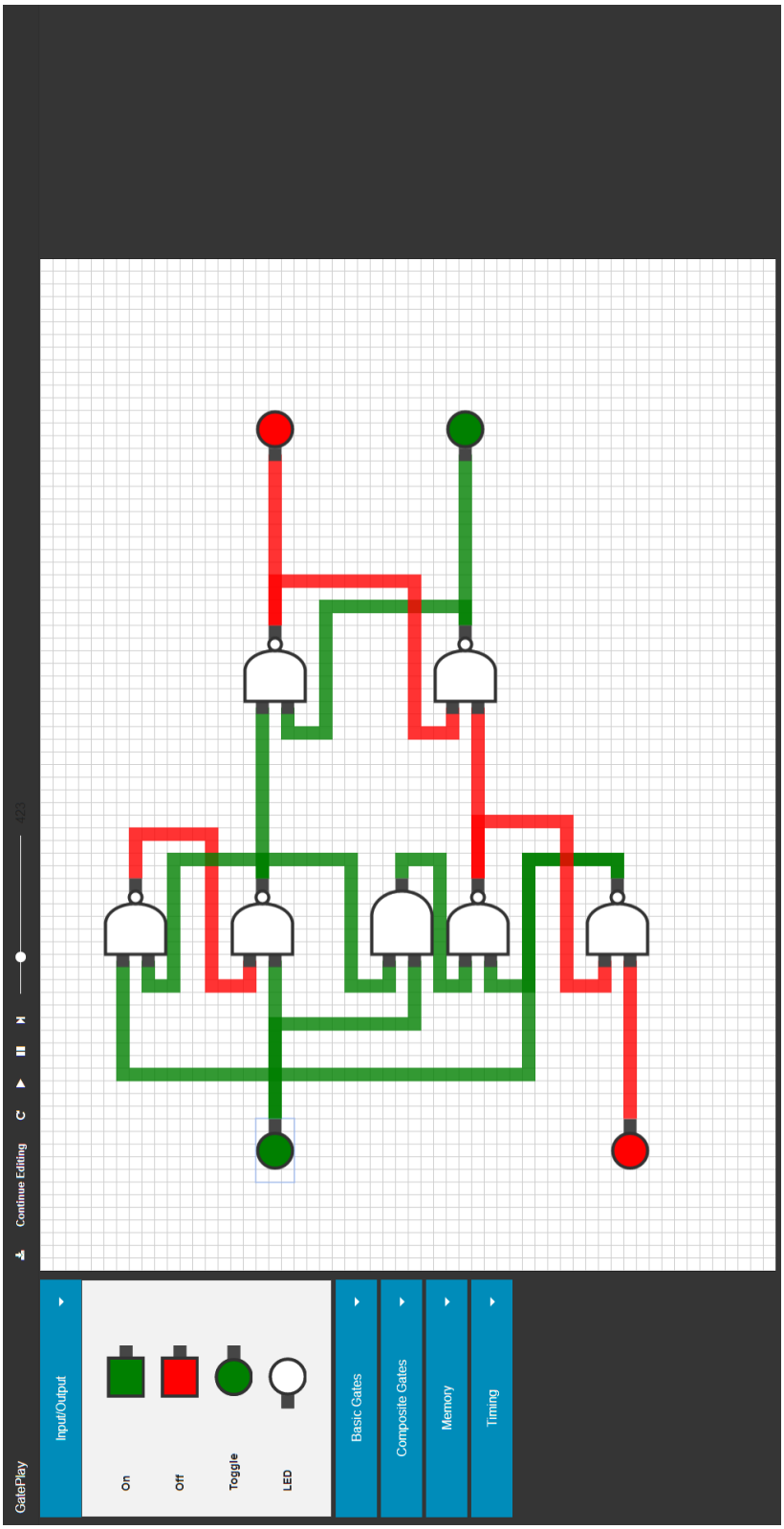

Figure 1.2: Simulating a D flip-flop

Chapter 2

Background

2.1 Logic Circuits

A *Boolean function* F is defined as

$$F : \{True, False\}^n \rightarrow \{True, False\}^m$$

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

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

A 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 (cite wikipedia).

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 and editing 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 clients 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 DOM tree and the styles of elements.

Chapter 3

Requirements

It was agreed 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 is 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. An explicit requirement was that the simulation of a ring oscillator (such as a ring made of an odd number of NOT gates) should show the circuit falling in to an

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.

- 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. The Evaluation Function also defines the propagation delay of the component, as discussed throughout this chapter.

Figure 4.1: Fields of a Component

- The **source component** the wire is leaving from
- The **output port** of the source component
- The **target component** 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

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 the following four values:

- The **source component** the event is propagating from
- The **output port** on the source component which has changed value

- The **event time** at which it is occurring
- 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 so called *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 do so is non-trivial and discussed in chapter 5.
- **External Components** like *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.

The heart of an event-based simulation is the event loop which processes all the events and generates new events.

1. **Fetch** 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, and add to the queue after the gate's delay

Figure 4.3: Event Loop Body

4.2.2 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.

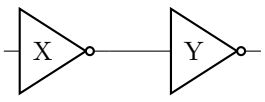

Figure 4.4: Two NOT Gates

When the event look body executes, the following actions occur:

1. Pop ev from the queue
2. Update the wire coming from X port 1 to $True$

3. A enters Y, so its input has changed. Apply Y's evaluation function to the new input, which returns $False$
4. Since Y's output changed, add the following new event to the queue: $(Y, 1, t + \delta, False)$ where δ is Y's gate delay

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

4.2.3 Culling

Events can sometimes be discarded without being processed by the entire event loop. 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.

4.2.4 Event Race Conditions

Consider the circuit shown in figure 4.5:

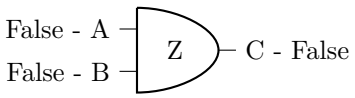

Figure 4.5: AND gate

Suppose there are two events in the event queue: $ev_1 = (X, 1, t, True)$ and $ev_2 = (Y, 1, t, True)$ where X port 1 is wired to A, and Y port 1 is wired to B. In other words, A and B both become $True$ at time t. Now consider what happens in the event loop at time t.

We handle each event independently, so suppose we handle ev_1 first: We set A to $True$, B is still thought to be $False$, and therefore we add the event $(Z, 1, t + \delta, False)$ 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 $(Z, 1, t + \delta, True)$ 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. If the event valued at $False$ 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. 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 — sets do not store duplicate values — of all affected components. In this example both events only affect Z, so the set will be $\{Z\}$. 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.

4.2.5 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 4.2.3) is critically important.

Also note that (if sensible data structures are used) event-based simulations are still performant in circuits with 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 2-Valued Simulation

4.3.1 Description

A simple simulation of logic circuits might use a 2-valued simulation. Each wire has one of two values: *True* or *False*. While it provides a reasonable model for logic circuits, it has some limitations described in this section.

4.3.2 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:

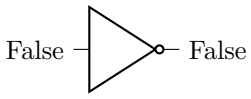

Figure 4.6: An inconsistently initialised NOT Gate

To have both A and B both be initialised to be $False$ is inconsistent with the logic of the circuit. The same would be true is 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 at first, however there is a more suitable solution introduced in section 4.4.

4.3.3 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 (from Wikipedia).

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. Our model of logic circuits does not consider these factors and therefore cannot make an informed estimation of the delay for each pass through a component.

For example, suppose the *NOT* gate shown figure 4.7 never has a propagation delay of less than δ_{min} nor a delay of greater than δ_{max}.

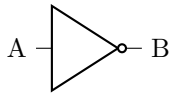

Figure 4.7: A *NOT* Gate

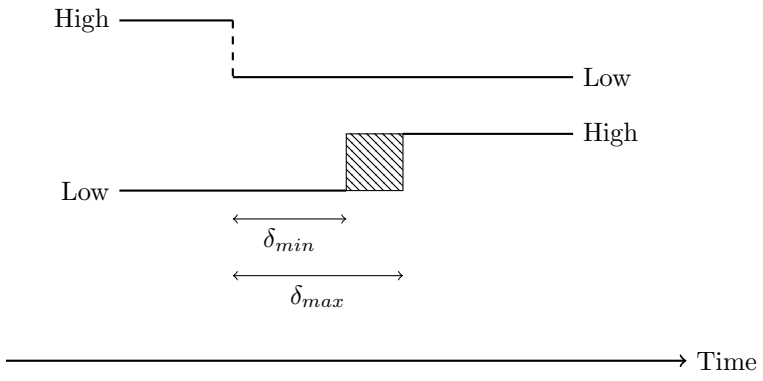

Figure 4.8: 2-valued Low-High transition

If we flip A's value from $True$ top $False$ the trace of the truth values through time is shown in figure 4.8. B's transition from Low to High 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 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 a run of this one 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 (see section 4.3) was to instead use a 3-valued simulation. A wire has one of three values: $True$, $False$, $Unknown$. A wire that is $Unknown$ may "in reality" be 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 4.3.2.

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 4.3.3.

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 a certain value for *all* possible combinations of gate delays.

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 delay respectively. A, B, and C are all $False$.

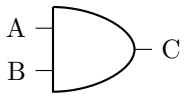

Figure 4.9: An AND Gate

Let circuit event e_1 set A to $True$ at time t, and event e_2 set B to $True$ at time $t+1$. 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 $e2$ we will add these 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 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 it 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

Make some general points here

5.1 Getting Started

5.1.1 Require.js

When the GatePlay url is visited, the only file downloaded is the main HTML file: index.html. The index then directs the user's browser to download the additional CSS and JavaScript files needed to use GatePlay.

A JavaScript script may use methods or variables defined in the global namespace, even if they were put there by another script, provided that the script has already been run.

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

Figure 5.1: Example use of Script tags

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 as added or removed, or sometimes if a file were modified.

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

```
1  // componentview.js
2
3  require([
4    // Declare the path of each file we require
5    "canvas/model/component"
6  ], function(Component) {
7    // Each included file is run, and we can give a name to whatever
         it returns if desired
8    var myComponent = new Component();
9    ...
10 });
```

Figure 5.2: An example file which uses Require.js

5.2 The Workbench

GatePlay's workbench is where we create and view circuits, and is implemented using an HTML Canvas element. A canvas is a blank slate which can have shapes and images drawn using the JavaScript API.

5.2.1 Fabric.js

An HTML Canvas only provides low level drawing tools. You are able to draw shapes and images on it, but there is no concept of persist objects on the canvas.

Fabric.js is a library which wraps HTML Canvases with an object model, allowing GatePlay to interact at the level of objects being added to, modified, and remove from the canvas, rather thana flat array of pixels.

Suppose I wanted to add a rectangle to a canvas, and then move it to a new location. Using Fabric.js this is two library calls (one to add a rectangle object and one to change the position property of the object). Using an HTML Canvas it is still one call to draw the rectangle, but moving it would require calculating what would be behind the rectangle, drawing that over the rectangle, and then re-drawing the rectangle at its new location. Fabric.js greatly reduces the amount of boilerplate code otherwise 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.

5.2.2 MVC with Backbone.js

Model-view-controller[1] is a design pattern to simply program development. GatePlay uses MVC in the implementation of the workbench. MVC applications contain three types of components:

- **Models** which store some part of the state of the application
- **Views** which display a representation of one of the models to the user

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

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

captionDefinition of a Wire model

- **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.

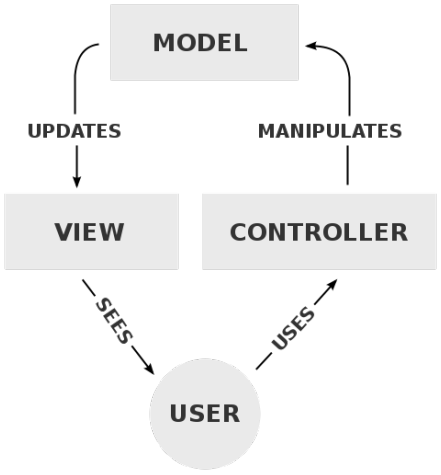

Figure 5.3: Interaction of MVC Components, from Wikipedia

Backbone.js[2] is a JavaScript library to reduce the amount of boilerplate code in developing MVC JavaScript applications.

One model is the Wire model as shown in figure 5.2.2. It includes a lot of the same information as the Wire objects used in the simulation, but also includes information regarding the *fixed points* of the wire.

We then create a view associated with the Wire model (figure 5.2.2).

[2]http://backbonejs.org

```
Backbone.View.extend({
    initialize: function(options) {
        // When the model is changed, update the view
        this.model.on("change:fixedPoints", this.render, this);
        this.model.on("change:truthValue", this._setWireColor, this
            );
    },

    render : function() {
        var model = this.model;

    // Using the model data we can now draw wires on the canvas
    },

    _setWireColor: function() {
        var truthValue = this.model.get("truthValue");

        // We can now re-render the wire with the new colour
    }
});
```

captionDefinition of a Wire view

5.2.3 Editing Mode

5.2.4 Running Mode

5.3 The Simulator

The implementation of GatePlay's simulator is a relatively straightforward implementation of the algorithms and ideas explained in section 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.

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

Figure 5.4: Implementation of *Blinker*'s evaluation function

```
function tick() {
  // For each 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);
  }

  // Event loop goes here
}
```

Figure 5.5:

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.

To implement this, a *Blinker* component's evaluation function (shown in figure 5.4 determines it truth value backed on the simulation clock.

However a *Blinker* will never be processed by the event loop, as no events will ever affect a *Blinker* (it has no inputs). Therefore 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 time 0 and 1, *False* events at time 2 and 3, and so on. We rely on culling to eliminate the replicated 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 5.5.

Note that the *blinkerEventQueue* is just a subset of the main event queue, and we could actually put this algorithm directly in 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 the drag-and-drop nature of the interface is an important part of that. 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

We first create an HTML Image element for each component on the left-bar. The matter of 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 Fabric.js.

jQueryUI[3] is a JavaScript library to ease the creation of interactive web applications like GatePlay. jQueryUI supports *Draggable* and *Droppable* interactions, which implement exactly what we want. The components in the left-bar are marked as Draggable and the main workbench is marked as Droppable, meaning components can now be dragged from the left-bar to the workbench.

To actually add the components to the workbench it is a matter of handling jQueryUI's *drop* event.

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 5.6

5.5.1 application.js

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

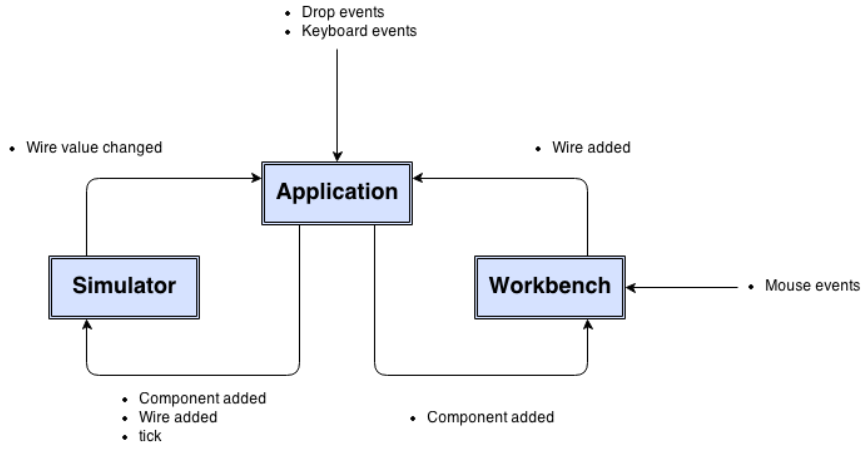

Figure 5.6: Information flow in GatePlay

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.

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

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 ONs, an OR, and an LED).

6.1.1 Ring Oscillator

A ring oscillator[2] can be implemented as an odd number of NOT gates arranged in a loop, as shown in figure 6.3.

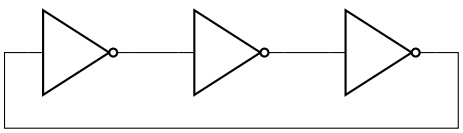

Figure 6.3: A ring oscillator

[1] http://qunit.com
[2] http://en.wikipedia.org/wiki/ring_oscillator

```
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

As discussed in section 4.3.3, the delay of components is not constant. Therefore the frequency of the oscillator — the sum of the gate delays of the components — is not constant. Therefore after some number of oscillators (precisely $\delta_{min}/(\delta_{max} - \delta_{min})$ 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 $Unknown$. I wrote a unit test checking that a 3-NOT ring oscillator does so: the code is identical to that in figure 6.2 except the circuit is 3 NOT gates, and the expected events

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[3] 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 acting as the user.

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

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

function circuitTest() {
  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);

  // The first two changes are wires 1 and 2 changing U -> T
  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;
  }

  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

6.3 Fail-Fast Methodology

Fail-fast is not a testing strategy, but because it helps in the development of bug free (or as close to bug free as possible) programs I will mention it here.

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

Chapter 7

Conclusions

Bibliography

[1] http://logic.ly.