Individual Report – Christopher Wilson

Project Delta - An Interactive FPGA Circuit Simulator

Developers:
Robert Duncan - rad55@cam.ac.uk
Justus Matthiesen - jm614@cam.ac.uk
DavidWeston - djw83@cam.ac.uk
Christopher Wilson - cw397@cam.ac.uk
Rubin Xu - rx201@cam.ac.uk
Client:
Steven Gilham - steven.gilham@citrix.com

Contents

Contribution to the Project 1
Drawing Components 1
Adding Components 1
Wires 2
Linking to the Simulation 2
Saving and Loading 2
Miscellaneous GUI Functions 3
Evidence of Contribution 3
Contribution of Other Group Members 4
Robert Duncan 4
Justus Matthiesen 4
David Weston 4
Rubin Xu 4

Contribution to the Project

My assigned role in the project was to work on the GUI, and in particular the circuit diagram. The idea was that the user would be able to drag components from a "toolbox" and drop them into the diagram, where they could then connect them up using wires. There were therefore four main areas I had to work on:

- Drawing components
- Adding components to the diagram
- Drawing and manual routing of wires
- Linking the diagram to the underlying circuit simulation
- Saving and loading of circuits
- Miscellaneous GUI functions such as undo, redo, copy, paste, etc.

Initially this seemed a very daunting task, but we found a very helpful library (JGraph) that would handle the rendering of components and maintain a suitable data structure for the diagram. This library undoubtedly saved a lot of development time, as I know longer had to implement a fairly complex diagram drawing program (with extra bits) from scratch. Most of my contribution has involved extending the JGraph classes in order to make them suitable for our applications.

Drawing Components

In JGraph each type of cell has a "model" class, a view class and a renderer class. Our electrical components they have conceptually different functions so I needed to subclass the "DefaultGraphCell" class for each different component. This was because each had a different number of inputs and ouputs, and would need to be represented differently when it came to simulating the graph.

Our components also look very different from each other. However they are essentially all being represented in the same way – as a circuit symbol. The easiest way to store to draw them on the diagram was to use the built in JGraph support for icons, and to have the view for each component hold a reference to the icon. Initially I did this by having a subclass of VertexView for each component, but then realised that it is only the icon that changes.

This led to combining all the view instances under one view class – DeltaComponentView. This holds references to each of the icons as constants, allowing them to be changed easily should the location be removed or renamed. To construct the right view for each component was a simple matter of checking the passed in "model" object when the view was created, and assigning the relevant icon.

Adding Components

According the specification, new components would be added by dragging and dropping them from a separate control panel. The layout of the control panel was David's responsibility, however I did need to provide a way of creating a suitable Transferable object whenever a component was dragged. I then needed to create a TransferHandler that could deal with the Transferable being dropped in the circuit diagram.

Again I found that JGraph has built in support for dropping, however I needed to subclass the adapter to provide a convenient way of letting the user choose which switch, LED or seven-segment display they wanted to represent.

To create the Transferable object was slightly more complicated. It involved creating a TransferHandler for the component panel that would detect which component was being dragged,

and create what is actually a small JGraph (i.e. a new circuit) to represent it. The ComponentTransferHandler class I created detects which component was being added based on a component "key" with the ComponentPanelLabel class. This was much easier than the alternative method of having a subclass of JLabel for every component in the panel.

Wires

This actually proved to be the most difficult part of the project. Drawing the wires was already implemented in JGraph, and there is even routing algorithm that will set up orthogonal wires ("edges" in JGraph terminology) between the two components. However, this routing algorithm does not allow the manual adding of control points, and if the algorithm is not used then the lines weren't orthogonal.

So essentially I had to implement manual wire routing from scratch. The route I took was to allow users to add as many control points as they liked to wires, and then to drag these to reposition different sections of the wire. This is not an easy task, and I decided that control points along a wire would alternate between "horizontal" and "vertical", as they controlled the either a horizontal or a vertical section. This is because intuitively it makes more sense to drag a wire from the centre of a section than from a corner.

The particularly difficult part is that whenever one section of an edge is moved, the control points around it must be realigned so that they are still in the middle of the relevant sections of wire. The basic algorithm for this is not too difficult, but it was clearly not something that was considered when the JGraph library was built, and I had to spend a long time understand how the JGraph rendering code worked in order to implement it correctly.

Linking to the Simulation

The simulation was being handled by Justus, who had found a fork of the JGraph library (JGraphT) that would give him a suitable data structure. Fortunately the developers had considered that people might like to link a JGraph graph to a JGraphT graph, and had provided a JGraphModelAdapter that handled the translation between them. With some minor alterations this suited us very well.

The only thing that had to be done from the GUI end was ensuring that the "userObject" of each component's "model" class pointed to the relevant JGraphT class, as this was how the adapter did the translation. This I implemented by creating a "replaceUserObject" method that was called when the component was initially constructed, and could also be called for things such as LEDs when the particular component being represented changed.

Saving and Loading

Java has the amazingly useful serialization feature, which allows a program to essentially save information about all its objects and their states to file. In most cases, just added the "implements Serializable" tag to all of my GUI diagram classes and Justus's simulation diagram classes was sufficient.

We only had one issue with getting serialization to work, and this was the SVG icons. So that the component icons would look good at different zoom levels, Rubin had used an existing SVG library to allow us to use vector-based images for them. Unfortunately this library didn't include the implements serialization tag, and obtaining a copy of the source code and adding it to potentially a large number of classes didn't seem like a viable option.

Instead, after relearning what I had been taught about serialization, I saw that you can customise what parts of an Object are actually serialized, and what is done with the information when an object is unserialized. It was then possible to implement a simple work around whereby the SVG icon's URI was serialized, but not the icon itself. On being unserialized the URI would be used to create a new copy of the icon. All of this functionality was contained within a wrapper class for the external SVGIcon (DeltaSVGIcon) class that can be found nested inside DeltaComponentView.

To perform the actual saving was therefore simply a case of serializing the DeltaGraph that held references to all the circuit's components. When a circuit is loaded, the DeltaGraph is unserialized from the file and the circuit panel is reloaded with that circuit. In order to prevent data on the current circuit being lost, David added a confirmation dialog box when trying to load a saved circuit.

Miscellaneous GUI Functions

The JGraph library was also most helpful in implementing many of these. One of the reasons the code for updating the graph initially seemed so complicated was that all the changes were being fed into "edits", which were recorded by the GraphUndoManager so that they could be easily undone and redone. Hence the undo and redo functionality was very easy to implement.

JGraph also has built in support for zooming, which I wrapped up in an Action class so that David could use it on the GUI. Cut, copy and paste were dealt with in the same way as drag and drop above, with Transferable objects and a GraphTransferHandler. The only extra work I had to make was in the Action classes, where I ensured wires that had been left dangling by a cut operation (or indeed a delete) were removed. Dangling wires were something we had decided not to allow.

Evidence of Contribution

Below is a list of the classes that I wrote myself. Some of the component classes were actually added by Robert so that I could concentrate on other things; however this was largely a case of extending the existing framework.

- org.delta.gui
  - ComponentPanelLabel
  - ComponentTransferHandler
  - Most of the Action classes – generally the ones that relate directly to the diagram like CopyAction, UndoActionk, etc.
- org.delta.gui.diagram
  - All classes except Dlip, Nand3Gate, Nor3Gate, RAM, ROM, RSLatch, SevenSegment and Switch
- org.delta.gui.diagram.images
  - I drew some of these icons and adjusted them to a uniform size. Others were contributed by Rubin and Robert, and the SVGs were converted to PNGs for the ComponentPanel by David

Although there are a lot of classes above, many of them are extensions of existing JGraph classes and only contain a few methods that needed to be overridden to implement custom functionality. The functionality of each class is described in the accompanying javadoc comments.

Contribution of Other Group Members

Robert Duncan

Robert was the obvious choice as project leader, and he has performed the role very well – as evidenced by the number of nagging emails I've had over the last few weeks! He also implemented the export to Verilog functionality and the transport between the board and the application. In the second half of the project he has helped out the rest of us by doing odd bits of work related to our own areas, like creating a number of my component classes and performing various GUI tweaks.

Justus Matthiesen

Justus's job was to simulate the circuit on the computer. This involved creating a suitable data structure to store the circuit and find a good simulation algorithm that could cope with both clocked and combinational logic. He quickly found the JGraphT library to represent the circuit (truth be told he also found the JGraph library that I used) and research simulation algorithms. I worked with him quite a bit to get the diagram to circuit translation working and he was very efficient and helpful.

David Weston

David was my fellow GUI designer. His task was basically to work on everything except the circuit diagram. Apart from having to find his way round the extensive Swing library, he has also used an external library to create the "concertina" effect on the component panel that we were looking for. His job required linking together all the parts of the project into one interface, which I think looks pretty good.

Rubin Xu

Rubin had a lot of work to do very early on in the project when we decided we would use a soft processor programmed into the DE2 board. We decided to use the JOP processor (a Java processor that somebody had created for a DE2 board as a PHD project), but it required a lot of work to get it working as we required. He also had to find a way of transmitting simulation information to and from the board over a USB cable. This was a key part of the project so had to be finished early, and since he has finished it he has done various other small tasks such as setting up the diagram to use SVG icons and writing the help file.