Crossroads
The ACM Student Magazine

Design of an Interactive Tutorial for Logic and Logical Circuits

by ***Jeremy Kindy***, ***John Shuping***, ***Patricia Yali Underhill*** and ***David John***

Introduction

The principles of propositional logic are essential tools for computer science students. An understanding of these principles has immediate applicability to software engineering, computer organization, and algorithm studies. Many beginning students have difficulty associating the boolean value of a logical statement with the input and output values of the corresponding logical circuit. This paper discusses the design and development of an interactive tutorial that assists in the understanding of these concepts. Perspectives of the beginning student and the team constructing the building blocks of the system are presented.

The tutorial is designed for two types of users. First are the introductory students who use the materials as part of their learning process. The tutorials are not intended to take the place of textbooks or instructors. They are simply another learning resource. In order to be effective, this resource must interact in a way that provides additional understanding of the material. The tutorials must run in a timely fashion and be readily accessible to the students. Second are the instructors who construct tutorials as supplements to their teaching. The tutorials must be flexible enough for faculty to make modifications.

Currently, there are several fairly advanced circuit development packages available. The use of these was deemed infeasible for several reasons:

1. many, if not all, of these packages require licenses for each computer running the software
2. installation and maintenance of the software is nontrivial
3. these packages are for circuit design, and are not meant to be a tutorial on

understanding the relationship between boolean algebra and logical circuits at an elementary level

Tutorial Overview--User's Perspective

Directing a web browser to **http://www.mthcsc.wfu.edu/~circuitproject** loads the home page for the project. The tutorial begins with an introduction to the elementary logical operations and logic gates. This includes statement definition and the negation, conjunction, disjunction, implication, and equivalence operators as shown in Figure 1. The exploration of the subject matter occurs at each individual's pace. Explanations, examples, and truth tables are provided, and ``Quick Quizzes'' can be taken along the way. Introductory computer science and discrete mathematics students benefit from this basic instructional module.

Statements

A statement is a sentence that is either true or false.

For example,

Four is divisible by two.
The moon is made of green cheese.
The number of widgets is less than sixty-six.

are all statements. Whereas,

How do you feel?

is not a statement.

It is often convenient when studying statements to represent them by variables. Usually, statements are represented by variables such as p and q. Then the following associations between the variables and that statments can be made:

p	Four is divisible by two.
q	The number of widgets is less than sixty-six.

Quick Quiz

Which of the following are statements?

1. Today is Monday.
2. I feel great today

1. Today is Monday.
2. I feel great today.
3. The federal budget supports nothing but good projects.
4. The federal budget is balanced.
5. The set of non-negative integers less than 232 each of which is a perfect square has cardinality no greater than the square of 14.
6. I earned an ``A'' in MTH117 and a ``B'' in CSC111.
7. I will earn an ``A'' in MTH117 and a ``B'' in CSC111.

Figure 1: Statement definition frame

In a following section, the simple gates AND, OR, and NOT are introduced as extensions of the logical operators conjunction, disjunction, and negation. For each gate, students interactively choose inputs and observe the results as a truth table is built dynamically (Figure 2). This is important for the beginning student because all circuits are constructed using these simple gates. There are designated colors for *true* and *false* that clearly mark the input and output values. The same color scheme is used in the more advanced sections where circuits are created. Since lengthy explanations are eliminated at this level, the tutorial is well-suited as a supplement to textbooks and lecture notes. A link to Java source code is included at the end of each tutorial page for those who are interested.

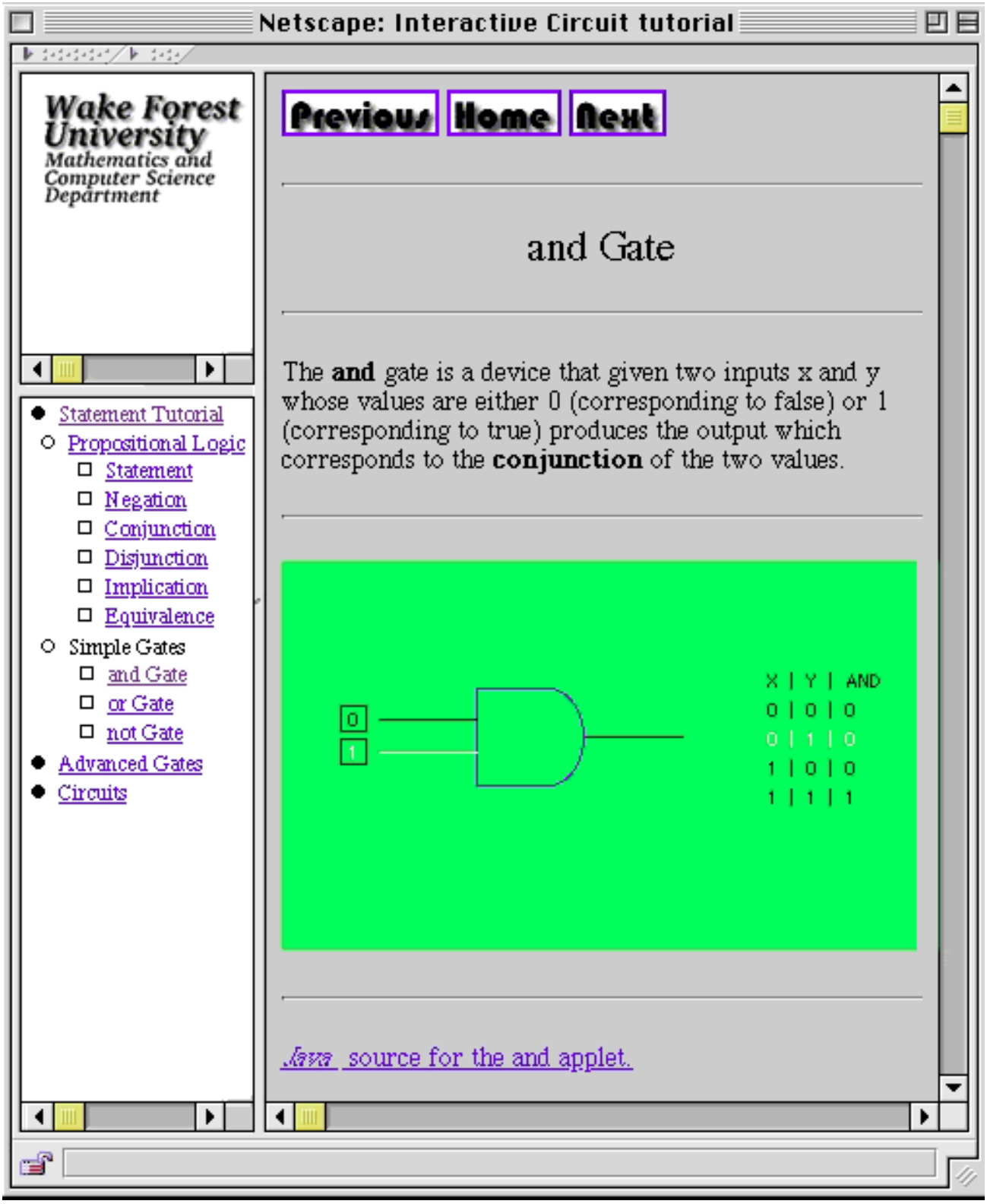

Figure 2: Browser view of the Tutorial, including navigation menu and interactive AND gate lesson.

Another section of the tutorial gives a brief lesson on binary addition. The half adder circuit is introduced to show how simple binary addition is implemented using XOR and AND gates (Figure 3). Again, the student can experiment with sets of inputs, and as before both the

circuit output and the truth table values are generated. In a later section, half adders are combined to make a full adder. This illustrates how simple gates and circuit components form the foundation for all digital computer hardware.

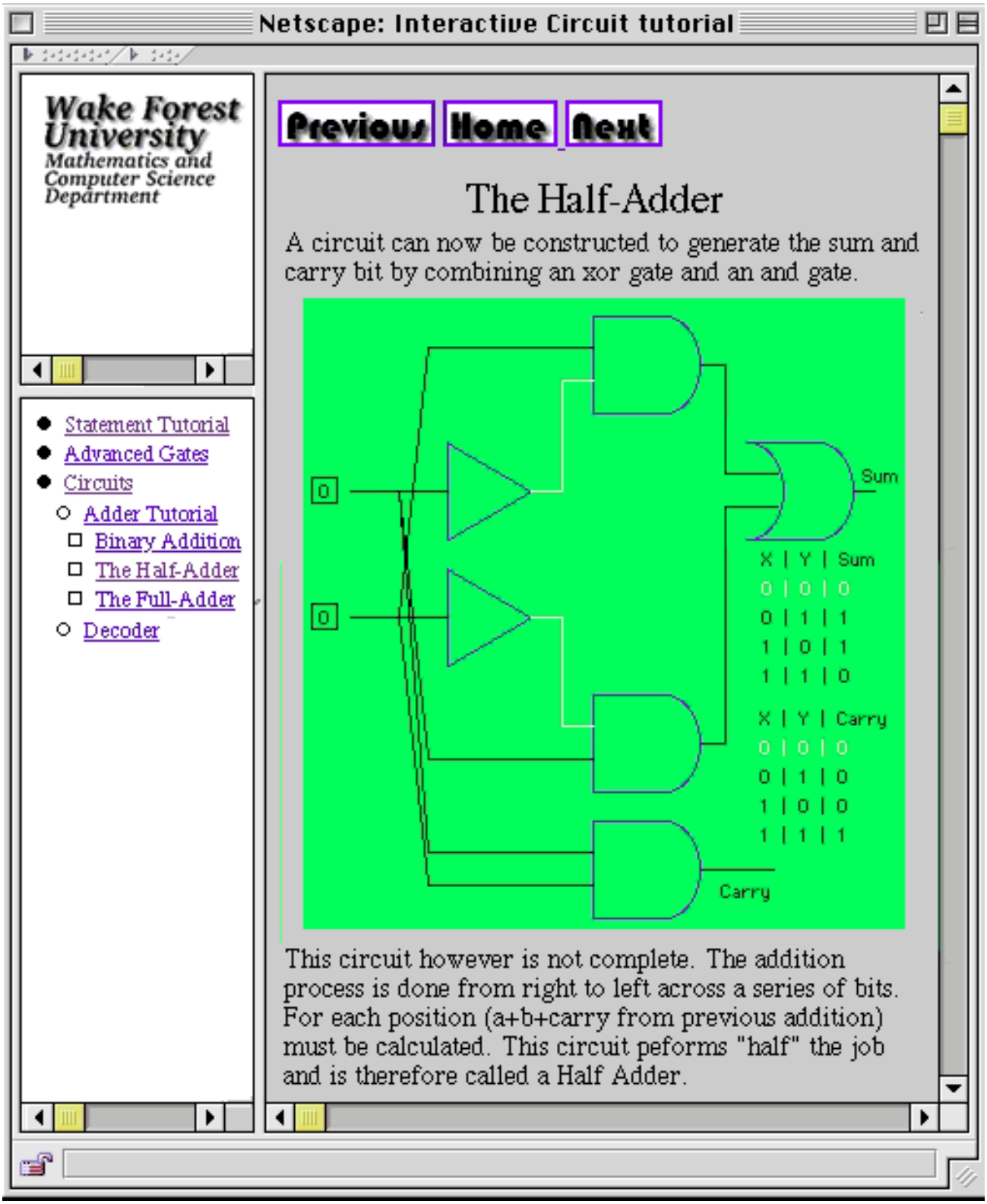

Figure 3: Browser view of the Half-Adder circuit.

The tutorial covers a wide range of topics. This allows students to study on different levels, depending on their background and past experience with the material. The tutorial is also useful in a variety of courses as content can easily be added.

Developmental Issues: Software Design

In order to best meet the needs of the different Wake Forest users, there are three primary goals of this project. First, build a set of reusable objects that represent basic logical circuit elements. Second, develop a reasonable mechanism for assembling the objects into tutorials. Third, assemble the tutorials in a coherent framework which facilitates access by students, forces a version control mechanism, and is easily maintained.

Fortunately, most of the issues related to machines, networks, and software are easily resolved at Wake Forest University. The University "Undergraduate Plan" specifies that all undergraduates and faculty are requisitioned a personal laptop computer [**1**]. Each of these nearly identical computers are configured with a standard software load, which is automatically upgraded by network servers.

Given these goals, Java was chosen as the programming language for the project. The main features guiding this decision were Java's object-oriented structure, relative ease with graphics, and portability. Java allows for individual gates and circuits to be created as either applets or applications. When the circuits and gates are packaged as applets, the tutorial is entirely web-based, with all of the object files stored on the web server. Java is part of the standard Wake Forest configuration, and is also part of the network browser.

The project was completed in two distinct phases. The first phase yielded a working set of libraries for creating gates and circuits, and the second involved refining the libraries and creating tutorials. Different teams worked on the two phases. Each phase is discussed below in two sections: a high-level overview of the phase followed by a more technical discussion.

Software Development: Phase 1

In the first phase of software design, logic gates were viewed as consisting of two parts: the set of input and output values, and the logical function of the gate that corresponds to the truth table. Using this logical division, two libraries were designed and implemented to serve as the basis for all logic gates and circuits: Gate and Connector.

In an initial prototype, the Gate and Connector classes were subclasses of the Applet class, and the AND, OR and NOT classes were subclasses of the Gate class. Objects in the AND,

OR and NOT classes were passed to the methods of the Connector class as Gate objects. This arrangement caused a very awkward situation. The Gate parameters could not be used naturally as AND, OR, or NOT objects. For this reason, the design was changed to avoid subclassing of the Gate class.

The Gate library provides a mechanism for creating, manipulating, and drawing any type of logic gate. The Connector library creates connectors between gates that both show the connectors on-screen and propagates the values from gate to gate within a circuit. Fairly early in the design the decision was made to have a fairly regular geometry for gates, which required the connectors to possess geometric flexibility in order to connect an output to inputs. Using these two libraries, a gate is created by writing a simple Java class. This class must create a gate, set its input values, create the connector, set the gate's starting position, and then instruct the gate and its connector to draw themselves.

Creating a class using the steps above yields a static gate on the screen with hard-coded input values. By adding functionality from Java's event handling system and a check-box at each input leg, the gate becomes interactive. When the value of a check-box is changed, an event handler notifies the gate of the change. The gate's input value is then set to the new value of the check-box. Next, the gate is redrawn by re-evaluating the values of each input, recalculating the output values, and propelling this value to any subsequent gates.

From the user's perspective, changing the value of a check-box causes the color and value of the corresponding input to change, and subsequently any inputs or outputs with new values are drawn in the proper color. This way, users see how their choices impact the output value of the gate or circuit.

After testing the libraries, interactive versions of each type of logic gate and some basic circuits were created. The circuits are created in exactly the same manner as the single gate, the difference being that more gates are instantiated and more connectors are drawn. These interactive gates and circuits are written as Java applets and then packaged into HTML files that show the truth table of the gate or circuit. These pages then are assembled into a web-based package. Feedback from a campus demonstration resulted in ideas for improvement in the software, which led the project into the second phase.

Technical Details of Phase 1

The Gate and Connector libraries are implemented as Java classes. The Gate class provides constructors and methods applicable to all gates, such as:

```
Gate(int kind, boolean val1, boolean val2);
```

```
void drawGate(Graphics g);
int getnumIN();
```

The above constructor uses the *int kind* argument to determine which type of gate (AND, OR, NOT, etc) is being created, and takes the gate's initial input values. In the Gate class, AND, OR, and NOT are declared as "public static final", allowing them to be used as symbolic constants for this parameter. The method *getnumIN()* returns the number of inputs of the calling gate, useful when using a combination of NOT and other gates. The Gate library also provides gate-specific methods which are usually called by a similar method available to any gate. This simplifies gate and circuit creation. The Connector library has three methods: one for drawing input connectors, another for output connectors, and a third for connectors between the output of one gate and the input of another. Simple and intuitive graphical representations of the circuits are used to focus on simple logical circuits rather than efficient circuit design.

As above, gates are written as applets or applications. The following is an example of a simple gate applet:

```
// Create the Gate and Connector objects
private Gate and = new Gate(Gate.AND, false, false);
private Connector andConnector = new Connector();

public void init() {
        ...
        // Set the gate's starting position on the screen
        and.setSTART(100,100);
}

public void paint(Graphics g) {

        // Draw the And gate
        and.drawGate(g);

        // Draw the 2 input Connectors
        andConnector.inputConn(g, and);

        // Draw the output Connector
        andConnector.outputConn(g, and);
}
```

Event Handling

A gate becomes interactive by adding check-boxes in front of each gate input and by ``listening'' for the check-boxes to be clicked. This is accomplished using the methods of *java.awt.Event*. When a check-box value is changed, the gate's *drawGate()* method determines the type of gate being drawn and then calls a gate-type specific *draw()* method. This method draws the actual gate. For example, the *drawAND()* method:

```
public void drawAND(Graphics g) {
   int width = 3*length/4;

   // set color for outline of the body and then draw it
   g.setColor(this.getGateColor());
   g.drawLine(xpos,ypos,xpos,ypos+length);
   g.drawLine(xpos,ypos,xpos+width,ypos);
   g.drawLine(xpos,ypos+length,xpos+width,ypos+length);
   g.drawArc(xpos+width/2,ypos,width,length,90,-180);

 };
```

After redrawing the gate, the connectors also must be redrawn. The methods of the Connector class redraw the connectors by first asking the gate for the new color of its legs and its position, and then drawing the connectors.

Coordinates and Gate Placement

The starting position of each gate (measured in pixels from the the upper left hand corner of the applet window) must be specified. The default position of each gate is at pixel coordinates (30,30). Failure to set the starting position of gates will cause them to appear on top of one another. To determine the position of a gate, the location and size of the surrounding gates and connectors must be computed. This can become complicated when creating a large, multi-gate circuit.

Software Development: Phase 2

The main focus of the project's second phase was to build tutorials using the classes Gate and Connector. A secondary objective was to determine the effectiveness of the the Gate and Connector class libraries. Many refinements in the initial design made the tutorial easier to use and understand, as well as more pleasing to the eye. Before these improvements, the placement of gates and check-boxes on the interface was erratic, and the truth tables were hard to read. The truth table and check-boxes needed to be integrated into the class system, and gate placement required simplification. Java allowed

for easy expansion of the existing framework to include the tables and check-boxes, without making significant changes to the Connector or Gate classes.

The first change implemented a grid system for gate layout. This allows the placement of gates on the interface using row and column numbers instead of pixels. The grid was created by adding a small method to the Gate class that takes the desired *x* and *y* grid coordinates as arguments. This grid starts at the upper left corner of the applet window. The addition of this method was the only modification to the Gate class.

The second major change aligned the check-boxes with the input legs of the gates. Initially, this was accomplished with a deprecated Java function named *reshape()*. Since this method would not be supported in the future, a custom check-box class named *cbx* was created. A black ``0'' was put in the box for *false*, and a white ``1'' for *true*, as those were the values used in the truth tables and they would be easily understood. The check-boxes automatically position themselves when provided with a Gate and an input.

The final significant change implemented an automatically updating truth table. Several types of graphical displays (mainly lines and/or boxes) were considered, but difficulty of placement created a problem with these solutions. Eventually, a working truth table was engineered using strings to correctly place values. The table automatically updates whenever an input is changed (when the *repaint()* method is called), with the new input settings in white, old (``visited'') input settings in black, and other (``unvisited'') input settings as invisible. The invisible strings contain only spaces.

As a consequence of these changes, the creation of a gate involves the instantiation of more objects. To create an interactive gate, one must create instances of gate, connector, truth table, and check-box classes. Then, the gate must be initialized into the grid system, mouse events occurring over check-boxes must be handled, and finally the gate and connectors must be drawn.

After the new truth table and check-boxes were tested, the previous applets were updated to use the new code, and the static HTML truth tables were removed. The modification of the HTML files necessitated a revision of the web pages. Also, a new color scheme was needed to make the gates contrast against the background. Green was chosen for the background, blue for the gates, black for ``off'' connectors, and white for ``on'' connectors. The table colors remained black and white.

Technical Details of Phase 2

Java's object-oriented structure simplified creation of the new grid system, check-boxes,

and truth tables. The grid system was implemented by adding a small method to the Gate class. The new method *StartGate()* takes two grid coordinates as arguments.

```
public void StartGate (int xpos, int ypos) {
        int newx, newy;

        // Translate the grid positions into pixel positions.
        // Each rectangle is 75x65 pixels.
        // (The grid is offset by 25 pixels from the top left of the screen)
        newx = (xpos * 75) + 25;
        newy = ypos * 65;
        this.setSTART(newx,newy);
}
```

The *cbx* class contains only *draw()* and *compare()* methods in addition to the basic constructs required for any Java class. When a mouse event occurs, the *compare()* method observes the location of the mouse event. If the mouse is clicked over a check-box, that check-box value and the corresponding gate's input are changed, and the applet is repainted.

There were problems related to the appearance of the check-box. Due to differing color settings in users' browsers, the ``x'' in activated check-boxes was sometimes invisible. To remedy this situation, the *cbx* class was modified to display a black ``0'' and white ``1'' of slightly larger size.

The main problem while designing the truth table was the varying number of inputs. This required the use of a different method for each number of inputs, which was accomplished through parameter overloading. The overloaded function compares the current settings to every possible combination of inputs, setting a string to each combination as it is reached. If a combination has not been reached, the string contains only spaces. The current combination is output in white, while the other ones, which have been set, are output in black. The strings are kept in an array of the size needed (although no more than two inputs have been implemented to date). Finally, the table is placed using the same grid system as the gates.

Here is an example of a simple gate written as an applet:

```
public class or extends Applet {
        private static Gate or1 = new Gate(Gate.OR, false, false);
        private Connector gateConn = new Connector();
```

```
        private Table truth = new Table(3,1,``OR Truth Table'',``OR'',4);
        private cbx or1_in0, or1_in1;

        public void init() {
                // Set the gate's position and create the checkboxes
                or1.StartGate(1,1);
                or1_in0 = new cbx(or1,0);
                or1_in1 = new cbx(or1,1);
        }

        // event handler
        public boolean mouseDown(Event evt, int x, int y) {
                ...
                compare and set input states
                ...
        }

        public void paint(Graphics g) {
                ...
                drawing functions
                ...
        }
};
```

Conclusion

The tutorial has undergone two major cycles of design and development. At this point, the only introductory tutorials developed for this package are the products of the development team. These materials have been field tested at two university technology expositions. Now the package is ready for use by other faculty with their students. Additional modules will be added to the existing ones, and the system will undergo further testing.

Java has simplified some tasks. It has been an excellent choice for providing straightforward control structures, mechanisms for an event driven system, and graphics support. The network browsers' ability to quickly load the object code and then begin execution is essential. Using compiled Java code with the browsers has been quite successful. Unfortunately, faculty who design lessons with this system must be able to construct very simple Java programs, as well as HTML. This is not a problem for our target faculty in computer science. On the other hand, if the faculty audience is extended, this could be a considerable problem. As discussed earlier, one large gate class was built, instead of a collection of smaller classes, one for each specific type of gate. Initially, there

was concern about the time required to load this class across the network, but transfer time has been negligible.

One point of concern for this project is the evolution of slightly different implementations of Java on different Web browsers. Even though the typical Wake Forest student will use the browser installed in the standard load, the typical computer science major is more likely to experiment with different browsers. This presents an interesting challenge in supporting the tutorials over a long period of time.

As a student programming design project, a couple of important lessons have been learned. First, the design and implementation of the software in two distinct phases worked quite well. In the first phase all attention was focused on gates and interfaces, and in the later phase those objects were used to build other objects. The second phase thoroughly tested the objects built in the first phase. Second, having different students responsible for the two phases was very productive. Only at the end of the project did the two students meet to look at the end result. This encouraged very frank evaluation of the two separate phases.

The end product is a simple interactive tutorial that can be conveniently accessed and used by students. Also, faculty can add and modify lessons in the collection in a relatively simple and direct fashion. This package was not designed to provide the sophistication of professional circuit design systems. The intent is to assist beginning computer science students in their mastery of critically important, yet fairly simple, ideas related to logic and logical circuits.

This must be considered a work in progress. The software design and implementation is complete. Now the product must be used by students in the beginning classes. Some care must be taken to quantitatively and effectively measure its effect. At this point the only evaluation of the interactive tutorial is anecdotal.

References

1. Brown, D. et al. *Plan for the Class of 2000: Final Report of the Program Planning Committee* Wake Forest University. Jan. 1995.
2. Cusick, G. *Java Logic Simulator* **http://www.cise.ufl.edu/~gjc/** (20 April 99).
3. Gajski, Daniel D. *Principles of Digital Design.* Prentice Hall Upper Saddle River, NJ. 1997.

4

Grand, Mark. *Java Language Reference.* O'Reilly Cambridge, Mass. 1997.

5

Ross, K.A., and Wright, C.R.B. *Discrete Mathematics, 4th ed.* Prentice Hall Upper Saddle River, NJ. 1999.

6

Sloan, M.E. *Computer Hardware and Organization, 2nd ed.* Science Research Associates Inc. Chicago. 1983.

7

Zukowski, John. *Java AWT Reference.* O'Reilly Cambridge, Mass. 1997.

Biography

Jeremy Kindy is a junior Computer Science major at Wake Forest University. His interests and activities include reading and cheerleading. He is the student who was responsible for Phase II of the project.

John Shuping graduated in May, 1999, as a Computer Science major at Wake Forest University. His interests include networks and computer security. He is now employed by Scient, working in San Francisco. He was responsible for Phase I of the project.

David John is an associate professor of Mathematics and Computer Science. His principle interests lie in the area of genetic algorithms. He is one of the co-advisors of the project.

Patricia Yali Underhill is an instructor of Computer Science. She is interested in the integration of Computer Science and Human Movement Studies. She is one of the co-advisors of the project.