Cross Platform Circuit Simulator

James W. Smith

BSc. Mathematics and Computing (Hons.)
The University of Bath
May 2007

Cross Platform Circuit Simulator

Submitted by: James W. Smith

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifcally acknowledged, it is the work of the author.

Signed:

Abstract

There are numerous circuit simulation products available today but the products which are clear and simple to use are often commercial, closed-source and platform dependent.

Current circuit simulators are analyzed to determine strengths and weaknesses, and a solution to their problems is discussed. This dissertation explores a new circuit simulation system combining the advantages of established educational tools such as *Crocodile Clips*, with the addition of cross-platform support and an open license.

Contents

List of Figures

Acknowledgments

Thanks to all those who helped with this project.

Dr. Russell Bradford
Project Supervisor

Pupils of Clarendon College, Trowbridge
Requirements Capture, Testing

Claire Burnley
Proof Reader

Chapter 1

Introduction

1.1 Circuit Simulation

Electronic circuits are important tools in everyday life and the understanding and development of new circuits and devices is important for the furthering and improvement of our technological society. Building and testing new circuits can often be an expensive, complicated and dangerous practice, so tools which allow for the simulation of circuits are essential for educational and development purposes.

1.2 Cross Platform

A platform is a combination of hardware and software used to run software applications. A platform can be described as simply as an operating system or computer architecture, or it could be the combination of both. For example, probably the most familiar platform is *Microsoft Windows* running on the *x86* architecture.

For a system to be *cross-platform*, it must be able to function on more than one computer architecture or operating system, and ideally as many as possible.

1.3 Problems With Existing Systems

There are numerous circuit simulation products available today but the products which are clear and simple to use are often commercial and closed-source, such as the educational software Crocodile Clips. This means that it is impossible expand or improve on these products. For example, if you wish to use a new type of component in your simulation and the product does not support that particular component, there is usually no way to add it in or create it yourself[1].

The other obvious problem with commercial products such as Crocodile Clips is the fact that often each copy of the product requires a software license. Commercial software licenses are often expensive and restrictive, and apart from the obvious financial cost of the license, which could be prohibitive to institutions such as schools, there are also management implications such as monitoring how many copies of the software are installed.

At present there are no commercial circuit simulation packages which are designed to run on operating systems other than Microsoft Windows. This can cause problems for institutions or individuals with established computing systems running alternative operating systems. The lack of a cross-platform solution hinders collaboration and increases overall system costs.

There are numerous free circuit simulation packages available, such as *SPICE*, but these are often designed with specialist applications in mind, such as integrated circuit modelling, or complicated analogue circuit analysis. Other packages choose to model purely digital electronic circuits and omit analogue components altogether. Most freely available circuit simulators suffer from poorly designed, unclear user interfaces, making them hard to use for new or occasional users.

1.4 Motivation

This project is a worthwhile undertaking because there is already demand within University of Bath for such a product. Other educational institutions such as schools could use this project for teaching technology courses.

1.5 Aims

It is the aim of this project to combine the merits of existing circuit simulation systems with the benefits of cross-platform accessibility, modularity, ease of use, and an open source license.

In order to do this, this project will set about creating an extensible circuit modelling framework for digital and analogue electronic circuits, accompanied by a clear and usable user-interface to be used as a view onto the circuit model. This will allow alternative user interfaces to be built for the circuit simulator.

1.6 Dissertation Structure

This paper now continues into a literature review in the next chapter, discussing in depth the merits and problems of current systems, as well as investigating the challenges associated with simulating digital and analogue circuits. It also looks at the considerations for developing a cross-platform system. In Chapter 3 the requirements of the system are formed, based on information obtained in the literature review and the discussion of the problem with users of current systems. The paper then moves on to discussing the design of the system in Chapter 4, before moving on to the analysis of the implementation and testing of the system in Chapter 5. The results of the testing process are then discussed in Chapter 6. The paper finished with a discussion of the completed project in Chapter 7, including recommendations for the future of the system.

Chapter 2

Literature Review

2.1 Problem Description

Many tools currently exist to aid in circuit simulation, but they each have limitations which make them undesirable for certain uses. Educational establishments such as schools and universities require circuit simulation tools which are both easy to use and learn, but powerful enough to build complex circuits. Commercial solutions which fit these requirements are often expensive and restrictive and apart from the obvious financial cost of the license, which could be prohibitive to institutions such as a university, there are also management implications such as monitoring how many copies of the software are installed.

At present there are no commercial circuit simulation packages which are designed to run on operating systems other than Microsoft Windows. This could cause problems for institutions or individuals with established computing systems running alternative operating systems. The lack of a cross-platform solution could hinder collaboration and increase overall system costs.

2.2 Existing Solutions

Various existing solutions are available in the field of circuit simulation, each designed for different target audiences and intended uses.

2.2.1 Digital Simulation

Due to the relative simplicity of simulating the behaviour of digital circuits, there are many software packages available which are able to perform these simulations. A popular and free example of this is the application *TKGate*. TKGate is an event driven digital circuit simulator with a graphical editor [2]. The source code for TKGate is freely available to download and change, which encourages community improvement and fast bug fixes [3].

Another popular digital simulation package is *Logisim*. Logisim is very similar in appearance and operation to TKGate and is also an open source product. Logisim is designed as an educational tool and therefore has a much simpler and clearer user interface [4]. This package also contains greater circuit reuse facilities, allowing a saved circuit to be used as a sub-circuit in a new design.

2.2.2 SPICE

SPICE is a computer simulation and modelling program used by engineers to mathematically predict the behavior of electronic circuits. Developed at the University of California at Berkeley, SPICE can be used to simulate circuits of almost all complexities [5].

SPICE, which stands for *Simulation Program with Integrated Circuits Emphasis* is one of the most popular circuit simulation software available and has many derived versions which attempt to specialise in certain areas. SPICE does not natively come with a graphical interface, and is designed to run at the command line using a text-based circuit input file, so is not suitable for educational use.

2.2.3 Crocodile Clips

Crocodile clips is a digital and analogue circuit simulator designed for educational use. It has a simple interface which is intended to be easy to learn for new users, yet flexible [6]. Older versions of crocodile clips which were purely used for circuit simulation are popular learning tools in educational establishments such as schools and universities. The University of Bath, for example, uses Crocodile Clips as a teaching aid for logic circuits in Computer Science courses.

2.2.4 Current Solution Limitations

Digital circuit simulators, while useful for teaching logic and building integrated circuits, have limited uses. Combined analogue and digital circuit simulators allow for more varied circuit construction and are of greater use in educational establishments.

Solutions such as SPICE are designed for accurate scientific simulation of circuits, and therefore are not designed with ease of use as a priority. The data provided by SPICE is optimised for accuracy and precision, but this is at a cost of speed. A suitable solution for an educational establishment would need to have a clear and responsive user interface, and fine precision is not of such an importance.

Commercial solutions such as Crocodile Clips provide a clear and simple user interface, but suffer from the drawbacks of being closed source [3]. Licensing of commercial products is also a problem for educational establishments. An ideal solution would be the simplicity and power of a Crocodile Clips like application but with a free, open source license.

Crocodile clips, along with other commercial solutions, also suffer from limited portability. Commercial solutions are mainly developed for the ubiquitous Microsoft Windows operating system. Large educational establishments such as universities often have UNIX servers storing and serving data and applications, so an ideal solution would be able to run on both Microsoft Windows and UNIX environments.

2.3 What is a Circuit?

An electrical circuit is an interconnection of electrical elements such as resistors, inductors, capacitors and switches which creates a closed loop, giving a return path for the current [7]. In this situation the circuit is known as analogue. All real world electrical circuits consist of analogue components. Components of analogue circuits have some effect on the current and/or voltage in the circuit.

More recently, the idea of a digital circuit has become more prevalent, in which there is no concept of current or voltage. Digital circuits in reality are made up of analogue components, but it is possible, and in fact convenient, to model them as purely digital. In this case the only possible state at any point of in the circuit is on or off [8].

It is also sometimes convenient to model circuits with both digital and analogue parts, these are known as “mixed-signal” circuits. When modelling such a circuit there is a transition between the analogue and the digital parts, where some voltage range, for example 3V-5V, is interpreted by the digital circuit as "on" and anything less as "off". Some models may take into account values higher than the range as damaging the circuit, while others take any non-zero voltage to mean "on".

A suitable solution would need to contain both analogue and digital circuit simulation. Mixed-signal circuit simulation would be ideal.

2.3.1 Components

The constituent parts of a circuit are known as *components* and can be broken down into various categories depending on their properties and uses:

Linear Components

Components in which the current increases in direct proportion to the applied voltage, in accordance with Ohm's law, are known as linear components or linear devices. Resistors and capacitors are classic examples of such components [9].

Non-Linear Components

Any component in which the current is not directly proportional to the applied voltage is known as a non-linear component. Non-linear devices each have their own non-linear equation which determines the relationship. Transistors and diodes are both non-linear components.

Output Components

An output component is a component which indicates the state of the circuit at a particular point by performing some kind of recognisable output. A typical example of such a component is a lamp, which could indicate the presence of current at a particular point in a circuit. Output components can either be linear or non-linear.

2.4 Circuit Analysis and Simulation

Circuit analysis is the process of determining the state of an electrical circuit at any given time. The state of a circuit normally means the current and voltage at every node in the circuit, although it can also mean the physical state of active components, such as lights and motors. A *node* 2.1 is defined

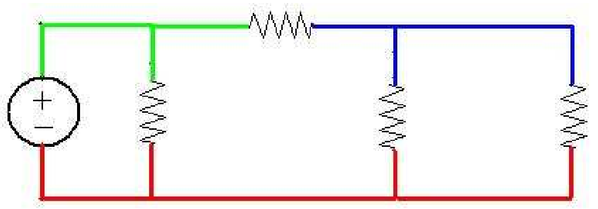

Figure 2.1: Each colour in the above circuit is a different node

to be any point on a circuit where the voltage is the same, and when modelling wires as having a resistance of zero, this usually means any join in the circuit or join of components.

It is often useful to know how a circuit will change over time, especially when dealing with components whose state depends on time, such as capacitors. This is done by the process of circuit simulation, which is the analysis of a circuit over a period of time. Circuit simulation represents how the physical circuit would behave.

2.4.1 Digital Circuits

Due to the nature of digital circuits, it is very easy to perform analyses on them. It is sufficient to simply follow the flow of “on” states through the circuit. Any well defined logic gates will have certain outputs depending on their inputs. These state tables need to be taken into account, as well as any digital timing components.

2.4.2 Analogue Circuits

Analogue circuit analysis requires much more computation as there are many possible values of voltage and current throughout the circuit. These voltage and current values are affected by the type and position of the components in the circuit. The methods used in the simulation of analogue circuits are discussed in depth in the following section.

2.5 Analogue Circuit Analysis Methods

There are various methods available to simulate analogue circuit behaviour. Certain methods are better at simulating certain situations and conditions, for example, some methods require only DC power sources be used or determine a circuit’s state once it stabilises.

2.5.1 Linear DC Analysis

Linear DC analysis is used to evaluate the DC currents and DC voltages for a linear time-invariant circuit [10]. Such a circuit would most often consist of simple a network of resistors and current and voltage sources. A circuit such as this would be "solved" by using *nodal analysis*.

The voltage and current values at each node in the circuit need to be determined by solving a series of equations based on the position of the components in the circuit and some elementary circuit laws, namely Kirchhoff's Current Law (KCL), Kirchhoff's Voltage Law (KVL) and Branch Constitutive Equations (BCE). [10]

Kirchhoff's circuit laws are a pair of laws that deal with the conservation of charge and energy in electrical circuits, and are used to determine the current and voltage at a certain node in a circuit. The Branch Constitutive Equations are a set of equations for components, such as resistors and voltage and current sources, which affect current and/or voltage at a branch.

Linear DC analysis would be of little use in realistic simulations as it does not allow for simulation of non-linear components such as diodes or transistors in circuits.

2.5.2 Non-Linear DC Analysis

Non-linear DC analysis expands on the methods used in linear DC analysis but allows for the inclusion of non-linear components such as diodes and transistors. Non-linear solution algorithms attempt to converge toward a solution to the system of nonlinear equations by iteratively developing and solving systems of linear algebraic equations [10]. Newton-Raphson iteration is the method most often used, and allows a linearised approximation of non-linear elements to be constructed.

Non-Linear DC Analysis is of little use in simulations where *real-time* information needs to be displayed, as the states at each node are calculated for a *steady state*. This means that it is only suitable for circuit analysis where the current and voltage values at nodes reach a steady state after a certain period of time. Calculations are performed as time tends to infinity in order to obtain the greatest accuracy. For *real-time* circuit analysis, where the components can be interacted with and output components monitored over time, another method of simulation is required.

2.5.3 Transient Analysis

In order to compute the behaviour of a circuit as a function of time, transient analysis is required. Transient analysis can be achieved by formulating and solving the series of differential equations which define the circuit's current and potential over time, but this is computationally very hard [12, 13]. An alternative way of doing this is to, like non-linear circuit analysis, transform the energy storing components, whose values depend on time, into linear models. Once this has been achieved nodal analysis can once again be performed.

Finding the analytic solution of a differential equation at discrete time points requires knowledge of calculus techniques which cannot easily be performed on a computer [13]. To get around this problem it is necessary to find an approximation to the analytical solution using numeric integration.

To transform an energy storage device such as a capacitor, first the current-voltage relationship of a capacitor needs to be obtained, for example:
$V = \frac{Q}{C}, I = \frac{dQ}{dt}$, then numeric integration needs to be applied to it. Once this result has been obtained it can then be used to develop a linear model of the capacitor to be used in nodal analysis.

2.5.4 Timing

When performing a circuit simulation and displaying the outputs to the user, the calculations to determine the voltage and current must be carried out many times a second to make sure that the simulation is accurate and representative. The *resolution* of the simulation is determined by how small the time slices are. For example, these calculations could be carried out ten times every second.

Determining how many time slices to make is determined mainly by the complexity of the calculations required. More complex calculations will require more time to complete, meaning fewer calculations can be performed per second, which in turn leads to a less accurate and less responsive simulation.

A suitable solution would require the time slice calculation delay not to be obvious to the user, so that the simulation appears smooth.

2.6 Solving Equations

The complexity of the equations depends largely on which method is used to calculate them. There are two main computational parts to simulating real-time circuits. The first part is to represent any non-linear components as linearised equivalents, using numeric integration. Secondly the nodal analysis must take place. Both of these parts require some complex calculations.

2.6.1 Matrix Manipulation and Solving

When performing a nodal analysis of a circuit, it is convenient to represent the system of equations representing the position of the node and the characteristics of the components in matrix form. For example:

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix}$$

Gaussian Elimination and LU Factorisation

Gaussian elimination is an algorithm for determining the solutions of a system of linear equations such as the system above. The algorithm takes an arbitrary matrix and reduces it to a simpler form using elementary matrix operations.

In transient analysis it is sometimes convenient for the right hand side of the equation above to remain constant, while changing the left hand side. In this case it is computationally less complex to use *LU factorisation*, which factorises a matrix into upper and lower triangular components, to aid in solving [11].

Sparse Matrices

Circuit elements are usually connected to four or fewer nodes, therefore the matrices describing circuits are generally *sparse*. *Sparse matrices* are matrices that contain a high proportion of zero entries. It is possible to make use of this sparsity to speed up circuit analysis. Dense matrix Gaussian elimination or LU factorisation typically has a complexity of order n^3, while sparse matrices typical of circuits the run time grows only at about $n^{1.5}$ [11]. It is therefore preferred to use a matrix solution method which maintains this sparsity, thereby speeding up calculations.

2.6.2 Numeric Integration

Energy storing devices such as capacitors are governed by equations representing the current or voltage of these devices over time. To allow such devices to be used in a nodal analysis, we need to integrate these terms to "move time forward". For example, we must integrate the current through capacitances [11].

There are various ways of performing this numeric integration computationally which are either more accurate or less complex. The most basic method is *Trapezoidal integration*, which is computationally very fast, but less accurate than slower methods such as the *Forward Euler* or *Backward Euler* methods.

2.7 Limiting Scope

It is important to consider the target user for such a circuit simulator when designing, as there are many features it is possible to make available which may or may not be required. For the purposes of this project, only simple "real-life" circuits need to be considered, ones which can be prototyped on a simple bread board. Because of this consideration certain complex simulation problems can be eliminated, such as ac power simulation and complex components such as inductors.

Certain circuit characteristics such as interference and signal degradation need also not be considered for a simple system. This reduces the complexity of the solution and the amount of computations required [10].

By limiting the scope of usage to perform "realistic" rather than extremely precise calculation, the computational complexity of the equations can also be reduced, without drastically affecting the realism of the simulation [11].

2.8 Circuit Description

To represent the circuit which is going to be simulated, a circuit description file is needed. This needs to detail all of the elements and devices in a circuit and how they are connected. For development and reuse purposes, this file should be human-readable but yet concise and easy for a computer to parse.

The circuit description file format used for SPICE simulations seems to adhere to both of these requirements, and is possibly suitable for use in a new solution.

It would also be possible to represent a circuit in an XML file, although this would be more complex and require an XML parsing library to be used.

2.9 Portability

Any solution would ideally be able to be built and run on most popular operating systems and environments, such as Microsoft Windows and UNIX. It is for this reason that portability needs to be considered.

2.9.1 Programming Language

The programming language an application is written in affects the type of system the application can be built and run on. Certain programming languages do not have compilers available on certain system.

The Java programming language uses a virtual machine, which means that a byte-compiled binary can be run on any system which has a Java virtual machine. Java suffers from inconsistent implementations and slow speed in certain situations, especially mathematical calculation.

C and C++ both have compilers available for a large number of environments, including Microsoft Windows on x86 processors, and UNIX on a large number of processors, and it well known to be a fast and reliable language [14].

2.9.2 Graphical Toolkit

The graphical toolkit used to produce the visual elements of an application can also limit the type of system the application can be built and run on. Certain toolkits are specifically designed for one system, such as the Microsoft Windows native toolkit, whilst others, such as GTK and Qt are designed to be used on many different operating systems and architectures. The Java graphical toolkits are portable to any system which can run a Java virtual machine, but again this suffers from inconsistencies across systems.

2.10 Designing the User Interface

A good user interface helps the user develop the correct mental model, so that it matches with the conceptual model developed by the designers. This can only be carried out by the involvement of the users to obtain the correct mental model. This will allow the user to understand their tasks and how to complete them. It is therefore important for potential users to be consulted during the design process.

There are many well established user interface guidelines available, but all build on the same core rules. The following list of considerations is from

Jakob Nielsen's *Usability Engineering*:

- Visibility of system status
- Match between system and the real world
- User control and freedom
- Consistency and standards
- Help Users recognize, diagnose and recover from error
- Recognition rather than recall
- Aesthetic and minimalist design
- Help and documentation

The GNOME human interface guidelines are another useful set of considerations which I shall be referring to during this project.

2.11 Summary

This review has discussed the key areas which required investigation in order to determine the true scope of this project. After discussing the problem, and outlining current solutions and their shortcomings, principles of circuits and circuit simulation were discussed and outlined. Analogue circuit simulation methods and techniques were discussed and their relationships to each other highlighted.

Important methods for solving linear equations were then outlined and compared and finally the scope of the project was clearly defined along with a discussion of the circuit data file format.

It is clear from the evidence presented in this review that a cross platform open source analogue and digital circuit simulator would be an ideal solution to the problem. The solution would use systems of linear equations to represent a non-linear transient system, and this system would be solved frequently over a real time frame.

Chapter 3

Requirements

The main aim of this project is to develop a circuit simulation system which is easy to use, and can run on many platforms. The project is aimed at being used in an educational establishment such as a school or university, as a replacement for tools such as Crocodile Clips when teaching.

3.1 User Requirements

A good source of ideas for features and requirements is to talk to actual users of current systems. The author of this document worked with a group of students from the *Clarendon College*, a secondary school in Trowbridge, to help capture requirements for the new circuit simulation system. The author is also a user of packages such as Crocodile Clips, having used them in both secondary school and University, and will contribute further ideas to the user requirements.

Familiarity and Consistency

Many of the users of this software will be familiar with existing circuit simulation packages such as Crocodile Clips. Indeed, Crocodile Clips itself is used by teachers at Clarendon College when teaching basic electronics, as part of the science courses, as well as at University of Bath to teach logic circuits as part of the Computer Science course.

A consistent look and feel enables users to apply their existing knowledge of their computing environment and other applications to understanding a new application. This not only allows users to become familiar with new applications more quickly, but also helps create a sense of comfort and trust in the overall environment [18].

Simple, Clear Interface

It is important that the interface does not contain irrelevant or confusing information when performing simple tasks. If it is necessary to include such information, it should be available on demand separately through extended functionality. Every extra piece of information or interface control competes with the truly relevant bits of information and distracts the user from important information [**?**].

Some students at Clarendon College mentioned that when using Crocodile Clips for the first time, they found it difficult to access the components they required. This is because Crocodile Clips uses a non-standard ”layered” toolbar, which changes the entire toolbar when an icon is clicked.

Accessibility

Certain users of the software might have physical disabilities, which could affect the way they use the program.

It is important to consider that users could be colour blind, which means that any visual cues which depend on colour may become useless to certain users.

Similarly, it is important to consider deaf users, and ensure that any audible cues are also represented visually.

It is difficult, and outside of the scope of this project, to consider users who are completely blind, as it would require extensive additions. It is possible, and important, to consider partially sighted users though when designing the system.

Easy to Install

Teachers and staff without great technical knowledge may be required to install the software on new machines. The program should be easy to install, with as little additional software dependencies as possible. This may be difficult to accomplish with a cross-platform program, but should be taken into consideration anyway. A teacher at Clarendon College stated:

> “If it is hard to install then I probably won’t bother”

This is obviously a very important requirement, and will be considered throughout the design process.

In order to ensure that the program *is* easy to install, any required dependencies should be installed along with the program, so that the installation is seamless.

Ready to Use

A key point which came from discussion with both students and teachers at Clarendon College, was that any new circuit simulator must be ready to use as soon as it is deployed.

In order to fulfill this requirement, the circuit simulator must have as many commonly used components as possible available immediately. Less common components can be provided as add-ons but, ideally, as long as the design is clear and uncluttered, it should be possible to have many built-in components.

Recognisable Symbols

As the circuit simulator will be used as a teaching aid, it is important that the circuit displayed on the screen looks similar to a circuit which might be found in a textbook. A teacher at Clarendon College mentioned:

> "If the kids can't match up what's on screen to what they are learning in the lessons, then the program will be useless."

It is therefore important to use internationally recognised symbols for components in the circuit.

Sessions

When working on a circuit in the simulator, it might be necessary to save the circuit design to a file so that it can be loaded again at a later time. Building in support for sessions into the program is therefore very important. This is highlighted by this comment from a teacher:

> "We may teach this sort of thing over a number of lessons"

To go along with the *open* and *save* functionality, and to mirror other programs, it would make sense to also have *new* document functionality, which creates a blank circuit.

Cross Platform

Of course, one of the major aims of this project is to produce a cross-platform product. This was not expressly mentioned by any of the possible users I talked to, but is important for people who might deploy the software.

All of the users at Clarendon College would use the software on *Microsoft Windows* based machines, whereas University of Bath would probably deploy the software on the UNIX servers. It is therefore important to consider possible cross-platform issues when designing the program.

Modularity

In order to ensure that the program can be added to in future, possibly by different developers than the author, it is important to consider the modularity of the software when designing.

By separating the simulation engine from the user interface, it will be possible to create additional user interfaces in future, for example, to cater to blind users, or more advanced users.

Ideally it should be possible to add additional electronic components in the future, so a modular component system would be well suited to such a system.

Open Source

Another major aim of this project is to make the source code of the software freely available. Making a project open source encourages further development and bug fixes from members of the open source community [3].

Again this functionality was not specifically requested by possible users, but the author feels that it would be a worthwhile requirement.

3.2 Scope

It is important to consider the target audience for such a circuit simulator when designing, as there are many features it is possible to make available which may or may not be required. For the purposes of this project, only simple "real-life" circuits need to be considered, ones which can be prototyped on a simple bread board. Because of this consideration certain complex simulation problems can be eliminated, such as AC power simulation and complex components such as inductors. Certain circuit characteristics such as interference and signal degradation need also not be considered for a simple system. This reduces the complexity of the solution and the amount of computations required [10].

A teacher at Clarendon College stated:

"The more components in there, the more complicated it will look."

As mentioned above, it is outside the scope of this project to cater specifically for blind users, as this would require many hours of extra research and work.

By ensuring modularity, as mentioned above, this ensures that although the scope of the project is limited, it can easily be extended in future to provide additional functionality.

3.3 Functional Requirements

This section outlines the functional requirements for this project, i.e. the requirements which define the inner workings of the software. These requirements have been built from the user requirements after consulting with probable users. They may be expanded on or evolve as the project develops.

I have indicated the priority of each requirement by using the words *must*, *should* and *may*.

1. The system *must* support creation and editing of digital circuit layouts using a simple graphical user interface.
2. The system *must* provide exact simulation of digital circuit behaviour.
3. The system *must* include basic functional digital components, including:
    - Logic gates
        - NOT, AND, OR, XOR
    - Switches
    - Simulated inputs
    - Simulated outputs
4. The system *should* support creation and editing of analogue circuit layouts using a simple graphical user interface.
5. The system *should* provide a precise, realistic simulation of analogue circuit behaviour, limited to simulation of:
    - Voltage
    - Current
    - Timing
    - Output approximation

This does not include simulation of:

- Temperatures
- Interference
- Alternating current

6. The system *should* include basic functional analogue components, including:
    - Switches
    - Current and Voltage Sources
    - Simulated outputs
        - Lights, Sounds, Motion
    - Core components
        - Resistors, Capacitors, Transistors, Diodes
7. The system *must* consider accessibility issues, such as colour-blindness.
8. The system *must* be easy to deploy and have minimal dependencies.
9. The system *must* use appropriate visual symbols to identify components and events. Components should be represented visually by internationally recognised symbols where available and appropriate.
10. The system *must* support save and restore of circuit layouts.
11. The system *must* run on at least Unix systems.
12. The system *should* run on other operating systems and architectures, including Microsoft Windows, and be functionally identical on all systems.
13. The circuit simulation *should* be provided as a separate layer to the user interface, allowing different user interfaces access to the simulation data in future.
14. The circuit simulation layer *should* be modular in nature, allowing for easy creation of new components or modelling techniques.
15. The system *may* provide a way for users to create or modify components.

3.4 Non-functional Requirements

This section outlines the non-functional requirements for this project, i.e. the requirements which determine the operation of the system and define how the software will look and feel. These requirements have been built from the user requirements after consulting with probable users. They may be expanded on or evolve as the project develops.

I have again indicated the priority of each requirement by using the words *must*, *should* and *may*.

1. The system *must* be easy to use for users with basic electronics knowledge and moderate computer knowledge.
2. The system *should* be modelled on existing, familiar educational tools, such as Crocodile Clips.

Chapter 4

Design

4.1 Tools and Technologies

4.1.1 Programming Languages

This section outlines the design decisions behind choosing a programming language for this project.

Perl

Perl is a scripting language which runs on many platforms, including Windows and Linux. It is easy to learn and light on dependencies and is used for many applications worldwide. Perl also has a large, well established user base, and comes installed by default on most Unix systems. Perl is more complicated to install, and has not been very actively maintained for Windows Systems [16]. Perl is known to be slightly faster than its peers, such as Python, although it still lags behind compiled languages in speed benchmarks [14]. The author of the project is only vaguely familiar with Perl and the language would have to be learned for the project.

Python

Python is an object-oriented scripting language which runs on Windows, Linux/Unix, Mac OS X and many more platforms[15]. Python was designed for writing utility scripts, and has grown to support all sorts of additional libraries, including graphical toolkits. Unfortunately Python performs poorly under most speed benchmarks [14]. This means it is not suitable for a low-latency program such as a circuit simulator. Additionally the author has little experience with the Python language, making it an unsuitable choice.

Java

Java is a byte-compiled programming language, for which the interpreter is available on many platforms, including Windows and Linux. Most systems do not come with Java already installed, and the Java system cannot be easily bundled with application software. It is also traditionally perceived as a slower language, which is backed up by benchmarks[14]. For this project, speed is very important, so Java may not be very suitable.

C

C is a very popular programming language because of its speed and ease of use. The dependencies required for a basic C program are installed on every Windows and Linux system, and extra dependencies can be easily bundled as libraries. Many developers are proficient in the use of C, including the author, meaning that development of C programs can continue long after a program's creation.

C++

C++ builds on the speed and strengths of C, while implementing an object oriented approach. Generally, this allows developers design and visualise their ideas in a clear way and reproduce them in their code. Object oriented design also allows for easy extensibility, meaning that other developers can continue developing existing projects easily. Dependencies for C++ are the same as a C system in most cases, unless more powerful features of the language are required, such as vectors, when the standard C++ library is needed. The author is also proficient in the use of C++.

Chosen Language

This project very much lends itself to an object-oriented approach, as the requirement to have a modular program suggests. Using an object-oriented approach also allows for the clear visualisation of certain elements of the project as classes. Since the author is proficient at using C++, and C++ is known to be a fast language, it seems like the ideal language for this project. C++ also allows for inheritance, for example, a *Component* could be specialised as both a *DigitalComponent* and an *AnalogueComponent*.

Using an object-oriented language, such as C++, also lends very well to user interface design, which will be a key part of this project.

4.1.2 Graphical Toolkit

This section outlines the design decisions behind choosing a graphical toolkit for building user interfaces in this project.

Native Toolkits

Both Windows and Linux (XWindows) have native graphical toolkits available which create programs with the appropriate look and feel of the operating system the program is being run on. As mentioned in the requirements, it is important for the user interface to be consistent with what the user expects.

Unfortunately writing user interfaces using native toolkits does not lend itself very well to cross platform development, as each native toolkit looks and behaves differently, and have vastly different APIs, meaning developing an entirely new user interface for each operating system.

GTK

GTK, which stands for the Gimp Toolkit, is one of the most popular graphical toolkits used in Linux and provides a clean look and feel, with a simple API. GTK is also available for Windows and, by default, GTK for Windows emulates the native look and feel of the Windows toolkits. It is also possible to bundle the GTK libraries with your application, making installation trivial. The author has some experience writing GTK applications.

GTK does not natively provide a C++ interface, although it is an inherently object-oriented toolkit. There is, however, a C++ wrapper for GTK called GTKmm which is also available for Windows and Linux, among other operating systems. Both GTK and GTKmm are available under the GPL license, which is an open source license.

QT

QT is another popular Linux graphical toolkit. QT is a larger library than GTK, but provides native C++ bindings. It is possible to bundle the QT libraries with your application making programs easy to deploy. In windows, QT can emulate the native look and feel of programs, although there have been many reported rendering issues and problems with interoperability. QT also comes under a dual license, which means to use it in an open source project, you must use an older version. The author has no experience with QT.

wxWindows

wxWindows attempts to solve the problem of cross platform user interface issues by providing a single API for many platforms, including Windows and Linux. wxWindows performs very similarly to GTK on both Windows and Linux, but has more dependencies. The author also has no experience with wxWindows.

Chosen Toolkit

Due to the ease of portability of the GTK platform, and the availability of a C++ wrapper (GTKmm), the author has chosen to use it as the graphical toolkit on this project. The author also has some experience using GTK which should ease development problems.

4.1.3 Circuit File Format

As there is a requirement to save and load the state of the circuit to and from a file, there needs to be some defined format to save the file in. In the literature review, the SPICE file format was discussed, but this does not take into account the positions of the components and wires displayed in the user interface. To be able to store this information, a different file format is required. There are two main options for a file format:

Binary Data

It is possible to simply serialise the component and wire widgets and write all their information to a file in binary format. This would mean serialise and deserialise functions would need to be written. This method is fast and requires no further dependencies, but results in non human readable files.

XML Data

The alternative to binary data files is to create an XML scheme for circuits, components and wires, and write XML serialise and deserialise functions. This would perform functionally identically to saving as a binary data file, but would result in human readable save files. For example:

```
<circuit>
    <component type="AND" x="45" y="45 />
    <component type="Input" x="65" y="25 />
</circuit>
```

The problem with the XML format is that in order to easy handle XML files an additional library *libxml* is required.

Chosen File Format

The benefits of the built in serialisation and human readability of XML outweigh the fact that an extra library must be included, therefore XML data has been chosen as the data file format. The extra library, libxml, is available for all platforms which this project is being considered.

4.1.4 Smart Pointers

Smart pointers are objects which store pointers to dynamically allocated (heap) objects. They behave much like built-in C++ pointers except that they automatically delete the object pointed to at the appropriate time. Smart pointers are particularly useful in the face of exceptions as they ensure proper destruction of dynamically allocated objects. They can also be used to keep track of dynamically allocated objects shared by multiple owners [17].

In order to make use of smart pointers in this project, and additional library is required, *Boost*. Boost is a peer-reviewed library of C++ functions and classes to increase a developer's productivity. It is free, is low on dependencies and is very fast. The author has decided to use various Boost libraries, including smart pointers in this project.

4.2 High-Level Design

As mentioned earlier, circuit simulation lends itself very much to an object oriented design. With regards to the requirements specification and literature review, one of the main requirements is for the system to be modular. For this system there will be two main parts, the *Simulator* and the *Interface*. The Simulator classes are important as they will be used by both the Simulator and the Interface.

The final design architecture for the Simulator part of the project is as follows:

4.2.1 Simulator Class

The Simulator object is the main object that performs the decisions in the circuit. It maintains the connections between Components as well as han-

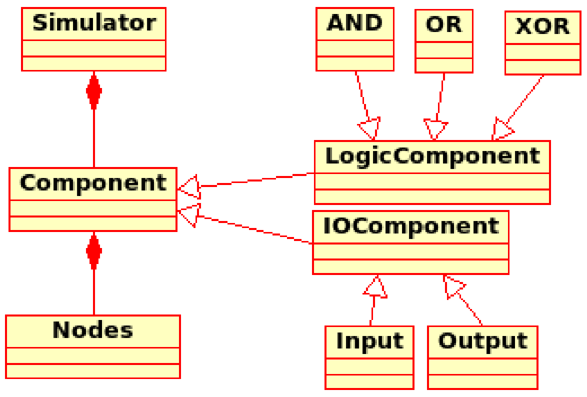

Figure 4.1: Class Architecture of Simulator System

dling the state changes throughout the circuit. The Simulator object contains a list of all the Components which belong to this Simulator.

4.2.2 Component Class

The Component class contains all of the information relevant to each Component. This includes storing a list of all the Nodes the component has. Each Component object also contains a virtual function Action(), which will define what sort of action is performed when the input Nodes change state on any derived classes.

4.2.3 Nodes Class

Each Nodes object contains a list of points to which Components can be connected. There will usually be two Nodes objects in each Component, one containing the input points, the other containing the output points. It is important to distinguish between input and output nodes in the Component class, as they each have distinct roles in the Simulator.

4.2.4 Specialisations

The above three classes would not be very useful on their own, so to create some actual usable Components, specialised classes are created. These classes inherit from Component and provide further details, such as the number of Input and Output Nodes, an Action() function which performs the component's action, and the reaction time of the Component (if any).

For example, the specialised classes shown in Figure 4.1 have been created, such as AND, OR, Input and Output, which themselves are specialisations of Component.

4.3 User Interface Design

The design of the user interface is a very important part of the project, as this is the part that most people will see.

The user interface will be a single document interface (SDI) allowing the creating and editing of only one circuit per instance of the application. This is because it is unlikely that users will want to edit multiple circuits at once.

4.3.1 Initial Window Design

The initial window design sketches were based around the interface for Crocodile Clips. This design would allow users familiar with Crocodile Clips to transition easily to this project.

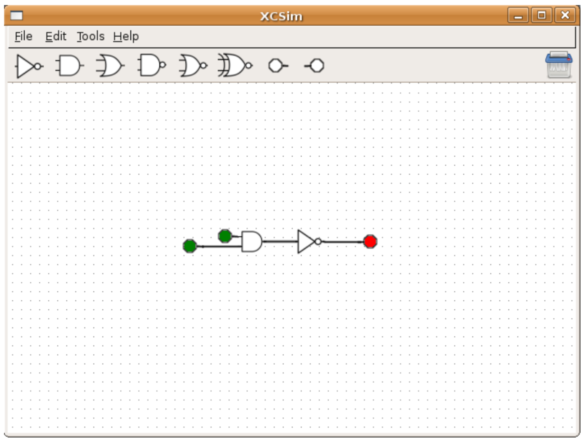

Figure 4.2: Initial Window Design

4.3.2 Revised Window Design

After reviewing the user requirements, especially the concerns about the “layered” toolbar used in Crocodile Clips, the main window design was changed to include a component browser pane, and more familiar toolbar buttons.

The design choices behind this new design are detailed below.

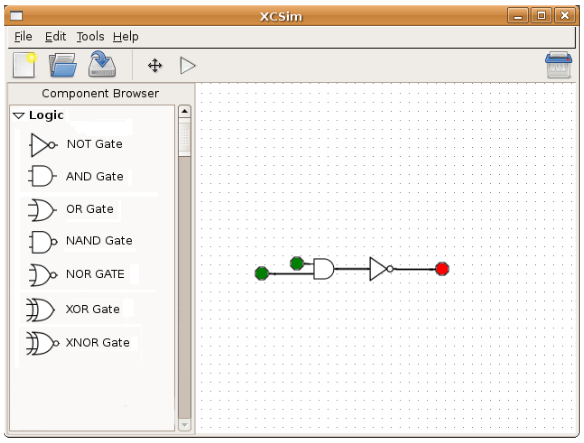

Figure 4.3: Revised Window Design

4.3.3 Interface Structure

The importance of consistency in a user interface was discussed in both the literature review and the requirements. The following sections identify design choices made to ensure consistency and ease of use in the interface.

Working State

This design follows a task-oriented approach, meaning that it should be clear which task is being performed, in this case either Simulation or Editing.

Fonts

One font style will be used to display text throughout the interface to enable consistency. Similar operations that require the use of text will utilise the same font style and size.

Menu Bar

The menu bar will contain all available tasks for the program, including New, Open and Save operations, and access to the program preferences. The menu bar will also allow for changing between the working states of the circuit simulator. The menu bar layout will be consistent with other programs.

Toolbar

The toolbar will contain the most frequently used tasks for the program, including New, Open and Save operations. The toolbar will also display and allow change of the working state of the circuit simulator. The toolbar layout will be consistent in look and feel with other programs.

Component Browser

In order to move away from the confusing toolbar layout of Crocodile Clips, electronic components were moved from the toolbar to a new element, known as the component browser. The component browser will be a scrollable, expandable list of components which are available for use within the circuit.

Circuit Workspace

The circuit workspace is the main area of the window where components can be added, manipulated and simulated. This will initially consist of a dotted grid to aid in circuit layout, and will be populated with components and wires as they are added by the user. Components and wires should “snap” to the grid to ensure good layout.

Chapter 5

Implementation

5.1 Development Tools

5.1.1 Compiler

To ensure cross-platform compatibility, as mandated in the requirements list, it was essential to choose a compiler which was available on many platforms. There are two main compilers which are available for many platforms, including Windows and Linux:

Intel Compiler

The Intel compiler is known to be fast and stable, but is available only under a proprietary license.

GCC/G++

The GCC and G++ compilers for C and C++ respectively are mature and fast. They are available for many platforms and are fully customisable. The author is also familiar with the GCC and G++ compilers.

Chosen Compiler

Due to the fact that the Intel compiler has a commercial license, the author shall be using the G++ compiler for this project.

5.1.2 Build Tools

To aid in the build process and to ensure that all dependencies are available before the project is built, the build utilities *autotools* shall be used. These tools include Autoheader, Automake and Autoconf, and allow for custom makefiles to be created which take into account the platform you are building on. The autotools are available for many platforms, including Unix, Windows and Mac OS X, so they are ideal for this project.

5.1.3 Source Code Editing

Having developed many applications before this one the author has experience with many Integrated Development Environments (IDEs) and text editors. An IDE includes support for not only writing the application source, but also debugging tools and revision control systems are also often included.

The author prefers to use separate debugging and source control systems, so the benefits of an IDE are less obvious. The ease of switching between project source files is of importance though, so various IDEs and source editors have been considered.

The author has previously used the IDEs Eclipse, Anjuta and KDevelop, as well as the text editors vim and gedit. Both Eclipse and KDevelop have very advanced functionality, although the interfaces can be inhibitive and confusing. Anjuta is less mature and is unstable.

The author chose to use the gedit text editor with the File Browser plugin, which allows for quick switching between source files. Project-wide searching and replacing will be done via the command line using tools such as grep and sed.

5.1.4 Revision Control

The source control system used while implementing the project was *CVS*. CVS provided all of the tools needed to comfortably write a project and roll back any possible errors made. *Subversion* was considered for the task, but the author was already familiar with CVS and as such, decided to use CVS. A CVS server was also already available to the author.

CVS roll back facility was never actually used during the project, however the author did utilise the CVS functionality of being able to view old revisions of source files, which was useful for bug fixing and ascertaining how the project had changed over time.

5.1.5 Debugging

For debugging purposes, the author preferred using the command line debugger gdb, rather than a graphical debugger. Gdb was used primarily to track down the locations of uncaught exceptions and segmentation faults. The package valgrind was also used in debugging, to aid in the locating of memory leaks. These two packages were chosen because the author was already familiar with them.

5.2 Simulator Classes

The first task undertaken during the project was to implement the classes for the simulator part of the project.

5.2.1 Nodes Class

The Nodes class defines the available pins on a component. There are usually two Nodes objects for each Component, one defining the input pins, the other describing the output pins. In this implementation the Nodes data values may only be a boolean value. This can be extended for analogue data. The current state of each pin is stored in a std::vector.

5.2.2 Component Class

The Component class is an abstract base class which is intended to be specialised by actual components.

The component objects contain a list of connections to other components. This data is stored in a ConnectionData object and placed in a std::vector. ConnectionData simple a std::multimap which links a Component object with an input/output pin.

Also, as mentioned in the design section, the Component object contains two Nodes, one for input and one for output.

The concrete component objects, such as AND, implement the Action() function. This function determines what values to set the component's output to depending on the values of the component's inputs. The Action() function is activated by a callback which is attached to the simulator. The simulator then calls Action() function indirectly via this callback when appropriate. To create these callbacks, the boost::function and boost::bind templates were used. The Action() function is attached to the simulator by calling the Enqueue() function of the Simulator object.

The concrete component objects also specify the number of inputs and outputs the component has, and the component's reaction time (if any). The reaction time of a component is the amount of time required before that particular component can affect its outputs.

5.2.3 Simulator Class

The Simulator class handles the changes of state of the various components and notifies the components of this state change. When a component experiences a state change on one of its inputs, the simulator calls the Action() callback for that component, ensuring the state is updated throughout the circuit.

There is a concept of cycles within the Simulator class, as each component can have a reaction time. The Simulator class keeps track of what cycle it is in to ensure that components with slow reaction times are not modified too soon.

The Simulator class maintains a list of components it is monitoring in a std::list, which is populated by the function InsertComponent(). It also maintains a list of callbacks to perform, along with the reaction time associated with that callback. These values are stored in a std::multimap.

5.2.4 Template Library Use

Template libraries are used frequently throughout the implementation of the simulation classes, as evidenced above.

Lists, vectors, pairs and multimaps from the Standard Template Library are used where appropriate to manage collections of objects.

Shared pointers are also frequently used due to their intelligent nature. These are part of the boost library.

5.3 User Interface Classes

To create the desired user interface, widgets needed to be created to represent the user interface features outlined in the design.

5.3.1 MainWindow Widget

The MainWindow widget class inherits from the GTK::Window class and describes the layout of the items in the circuit simulator window. It also connects any actions performed on the UI to functions.

The MainWindow consists of a menu bar, a toolbar, the component browser, the circuit workspace, and a status bar. Each of these items are constructed and placed in desired way using a series of layout boxes.

The toolbar and menu bar of the MainWindow are constructed using what is known as an action group. An action group is a GTK item which defines all possible actions which could be in a toolbar or menu bar. The required actions are then placed into the relevant bar by defining the layout in XML.

All icons used in the toolbar and menu bars are from the GTK stock icons, and they seem to portray their actions clearly. The working state icons on the toolbar are implemented as toggle buttons, meaning that it is always clear what the current state of the system is.

The component browser and the circuit workspace are displayed in what is known as a "paned view". This means that the width of the component browser and circuit workspace can be adjusted by clicking and dragging on the line between them.

The MainWindow class handles any click events that occur in the circuit workspace, but these events are passed on to the Workspace class once they have been classified, depending on the state of the system.

5.3.2 ComponentBrowser Widget

The ComponentBrowser widget class inherits from the GTK::TreeView class, and describes the layout of the component browser section of the screen. The component browser lists all available components in a tree list structure, allowing components to be easily grouped by type. The component browser section is contained within a scrollable widget, meaning that all components are accessible even with smaller screen sizes.

Clicking on a component in the component browser makes the component available for drawing, so that when the circuit workspace is next clicked, the selected component is placed in the desired place.

5.3.3 Workspace Widget

The Workspace widget class inherits from the GTK::Layout class, and describes the layout and appearance of the circuit workspace section of the user interface. The Workspace widget instantiates a Simulator object and is the main interaction point between the user interface and the simulation classes.

Any clicks received by the MainWindow widget in the circuit workspace area are classified and then sent to the relevant function in the Workspace class.

If the user is not in simulation mode, and clicks on the circuit workspace to add a component, this is passed to the InsertComponent() function, similarly there are EraseComponent(), ConnectLClick(), ConnectRClick(), EditLClick() and SimulateClick() functions which each handle workspace clicks.

The Workspace widget is also responsible for maintaining the dotted grid, and drawing component diagrams and wires. The functionality of the Workspace widget class is broken down below:

Adding Components

When the user selects a component in the component browser, then clicks in the circuit workspace, the MainWindow widget recognises the click and attempts to determine which component is selected. The MainWindow widget first looks at the id of the item in the list, then builds a ComponentWidget of the correct type. The ComponentWidget contains the graphical representation of the component, along with a pointer to an actual Component object of the correct type. The ComponentWidget is then passed to the Workspace using the InsertComponent() function.

The Workspace then checks to see if there are any existing components or wires at the clicked location, using the TestLandingSite() function. If the area is clear, then the actual Component object is added to the Simulation object, and the component is drawn.

Deleting Components

When the user is in delete mode and clicks on the circuit workspace, the MainWindow widget recognises the click and passes it to the Workspace class using the function EraseComponent(). The Workspace then determines if the user has clicked on a component, and calls RemoveComponent() on the Simulator if necessary.

Moving Components

When the user is in edit mode and clicks and drags on the circuit workspace, the MainWindow widget recognises the click and passes it to the Workspace class using the function MoveComponent(). The Workspace then determines if the user has clicked on a component, and calls StartComponentDrag() if necessary. The start and end coordinates of the drag are stored in the Workspace.

Connecting Components

When the user begins drawing wires on the circuit workspace, the MainWindow widget recognises the click and passes it to the Workspace class using the function ConnectLClick().

Each input or output pin on a component is stored in the Workspace's Hook. If the click joined two Hooks together, then the Workspace calls the Component's ConnectTo() function, making a connection in the simulator.

If the click was just on an empty area of the circuit workspace, the Workspace creates a new point on the PointMap, which stores the wires. Joining wires to wires connects them in the PointMap, then if these wires are connected to a Hook at either end the ConnectTo() function is called.

Interacting With Components

If the circuit simulator is in Simulate mode and clicks on the circuit workspace, the MainWindow widget recognises the click and passes it to the Workspace class using the function SimulateClick(). The Workspace then determines if the user has clicked on an input component, and if so toggles the state of the input on the Component object.

Building Component Symbols

As all electronic component symbols are designed to be easy to draw, it is also easy to construct them using vectors on a computer. Component images are built using vector commands, and many of these vector commands can be reused between components. For example, the symbol for an AND gate is almost identical to the symbol for a NAND gate.

The symbol images are constructed using the classes derived from the ComponentFactory class. For example, the ANDWidget is derived from the GateWidget, which in turn is derived from the ComponentWidget which is constructed using a ComponentFactory.

Additionally there is support for loading images in from a file, but this is not currently used in the project.

5.3.4 Utility Classes

There are various utility classes which are used in these widgets, some of them have been outlined above.

Chapter 6

Testing and Results

Testing is one of the more important aspects of this project as helps determine whether the project has passed the requirements and is fit for use in its intended environment.

6.1 Methods Employed

Various testing methods were employed in this phase. They are outlined below:

6.1.1 Author Testing

The project was initially tested by the author to ensure that it acted as it should do. There are two key "layers" to the project: the Simulation layer and the user interface layer. The user interface layer accesses all of the functionality available in the Simulator layer, so it is sufficient to thoroughly test the user interface layer.

The method that was employed to test this project was to use the circuit simulator as it was intended. During the implementation stage of the project, preliminary testing was done on as much of the new code as possible, eliminating most of the the bugs while the code was still fresh in the author's mind.

Accuracy

To test the accuracy of the simulated circuits, various test circuits were built, and compared against known values. While testing in this manner a bug was found where the NAND gate would act like an AND gate, causing the simulation to be inaccurate. Further testing using complicated circuits yielded no further problems.

Robustness and Speed

To test the robustness and speed of the simulated circuits, a large test circuit was created using 50 logic gates and various inputs and outputs. The program had no problems rendering the components and when simulating, the changed input values propagated through the circuit seemingly instantly.

6.1.2 Build Testing

As a cross-platform project, it is important that the software builds under environments other than the author's machine. A minimal copy of the project was produced which contained all the source files and the minimum additional files required to start a build environment. Namely, the configure.in file and the Makefile.am files. This was then packaged up into a compressed tar file ready for distribution.

The build testing initially took place on the author's machine, which has an x64 AMD processor, and is running Ubuntu Linux. The minimal copy was then extracted into an empty directory on the author's machine, and the autotools were executed (see appendix A). The configure script was then run without error, and finally the project was built using make.

Alternative Platform

The minimal packaged copy of the software was then placed on a CD-ROM and transferred to a secondary machine. The secondary machine had an x86 processor and was running FreeBSD version 5.0. The autotools where executed, followed by the configure script. This time the configure script halted, as expected, complaining of missing libraries. After installing GTKmm and its dependencies the configure script was executed again, this time completing without error. The project was then built with make, but failed soon after attempting to build.

The failure was down to the fact that the compiler was expecting the boost libraries but they were not installed. This should have been detected by the configure script.

6.1.3 User Testing

In order to obtain real feedback on the way the program looks and performs, a user testing session was organised with a group of Year 11 science students and their science teacher at Clarendon College school.

The circuit simulator application was installed and loaded on a laptop running Linux, and was given to the students in turn to test with. Each student spent roughly 5 minutes with the system, and the teacher roughly 10 minutes.

Feedback from the students was generally positive. All students were able to create and simulate a simple circuit. Some students commented on the ease of use.

Problems arose when attempting to change input states in the simulation, as some students had difficulty determining the state of the inputs.

Further problems arose with some students switching between edit and simulate modes in the user interface. Some students also had problems when drawing wires in edit mode, as it led to the components being moved around.

6.2 Results

As a result of the various test procedures employed, three major problems were identified:

6.2.1 NAND Simulation Error

The NAND component was incorrectly simulating as an AND gate. This was due to a logic error in the BuiltInComponents class.

This was easily corrected, and the NAND gate now acts as it should.

6.2.2 Build Failure

The project would fail to build on a clean system due to missing boost libraries. This problem should have been detected by the configure script but was not, leading to compile errors.

This was corrected by adding boost as a dependent library in the configure.in script. Now if boost is not present, the user is informed of this fact.

6.2.3 Input States

Some users had trouble easily ascertaining which state the input components were in. The values of 1 or 0 are displayed on the input components but it is not immediately clear.

By adding colour values to the input components it is now possible to see at a glance what state the component is in. Green was chosen to represent 1 and red was chosen for 0. It is important to note that these changes will not be of any use for some colour blind users.

6.2.4 Interface Context

Some users had trouble switching between interface contexts. Other users dragged components around when they were trying to draw wires. These problems become evident due to the fact that many different actions can take place in edit mode.

After discussing this problem with the students and the teacher it was decided to add further contexts to the user interface. The contexts now consist of Add Mode, Delete Mode, Wire Mode, Move Mode and Simulate Mode. The final window design after these changes can be seen on Figure 6.1:

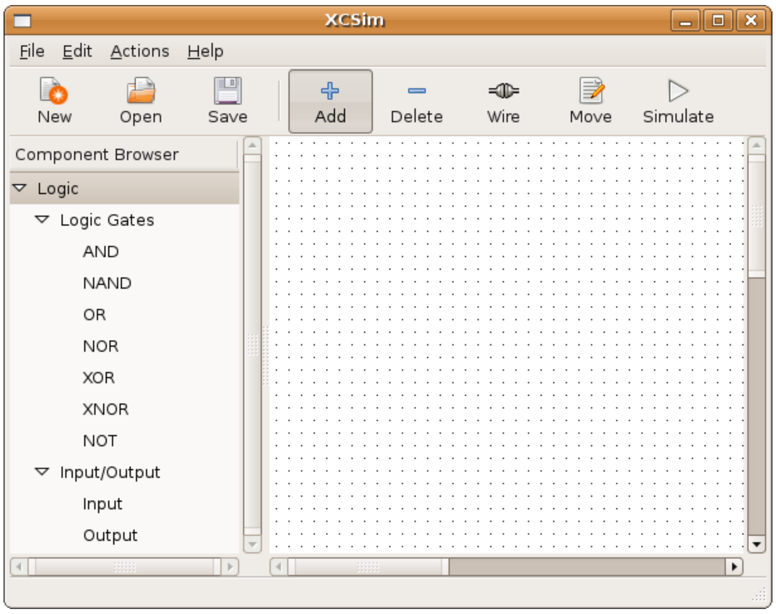

Figure 6.1: Final Window Design After Testing

Chapter 7

Conclusion

7.1 Requirements

When evaluating the project against the functional and non-functional requirements, it is evident that most of these requirements have been met.

Only one of the requirements classed as a "*must*" has not been implemented. The save and restore state feature, functional requirement 10, was not implemented due to the time constraints involved in learning and working with the libxml library. This functionality could be added in future into the Workspace class, as all of the required information is available. A developer more experienced with libxml and XML in general could implement the save and restore functionality with just two functions.

The analogue circuit simulation features listed as functional requirements 4, 5 and 6 have not been implemented, also due to time constraints. This was anticipated at the start of the project, and the design of the current system allows for analogue simulation to be added at a later date. Analogue simulation would have been ideal for use in an educational tool, especially in schools, so this functionality might be added as an extension.

Functional requirement 12, the requirement to run on other operating systems including Microsoft Windows, was not completed, although the system does compile and run on Linux and FreeBSD. Sufficient time was not allocated for the building of the system on Microsoft Windows, as this requires setting up a particular build environment. Given the right build environment, the author believes that the project would build without any significant problems on the Microsoft Windows platform.

Additionally, the final functional requirement, 15, was not completed. The requirement to provide a way for users to create components was listed as "*may*" and was listed as a possible extension if there was time left after completing the rest of the project.

7.2 Positive Aspects

Most of the project went surprisingly well. The program produced is functionally identical to the digital simulation in Crocodile Clips and, with the additions mentioned above, could be used for teaching logic in educational establishments.

Any bugs of significance that were found during testing were fixed. There are a few known bugs still within the system, but these were not considered major enough by the project author.

The speed of simulation, even with large logic circuits, is very impressive, certainly on par with similar products.

The code will be released under an open source license, which was one of the main aims of the project. Hopefully this will spur further development.

The project was also successfully designed in a modular fashion, another major aim. This means that functionality which is not available at the moment, can easily be added at a later date.

7.3 Problem Areas

The time management for the project, although adequate, could have been more efficient. Although development was not left to the last minute, the majority of the project was completed within the second half of the time frame. This was not an efficient use of time, although it did mean that no aspects of the project were forgotten.

The design stage of the project seemed thorough at the time, but the problem of unclear user interface modes, raised during user testing, meant that large portions of the user interface had to be changed in order to implement the new modes. If this problem was captured during the design phase then more time would have been available to work on additional features of the project.

In the project plan, the digital parts of the system were to be constructed and tested before the analogue parts, to ensure that if time was short, the analogue parts could be dropped and added later. This meant that when designing the simulator parts of the project, the possibility of analogue extensions had to be considered. This was a time consuming process, and meant that there are areas of the simulator design which do not currently have a use. Given more time, it would have been ideal to develop and design both the analogue and digital parts of the simulator together.

During the testing phase of the project, thorough testing was only carried out on the author's machine. This means that the program could act slightly differently on other machines, although this was not evident initially from the user testing. Ideally testing would have been carried out on a number of different systems.

7.4 Future Work

There are numerous possible future additions to this project, as the modular nature of the program allows for additional features to be easily added.

As mentioned earlier, analogue circuit simulation was not implemented in the project. An interesting future project would be to add this analogue support in to the simulator. If analogue simulation support was included, built in analogue components would also need to be added.

A crucial and very useful addition would be the save and restore feature, which has not currently been implemented. Ideally this would be implemented by someone with libxml and XML experience, or someone willing to learn.

The modular nature of the project allows for alternative user interfaces to be created. These may allow for further information display, such as the graphing of circuit states. Additional user interfaces could also cater for disabled users, such as blind users, by implementing screen reading technology.

Although the software seems to run very fast, there has been no formal optimisation. It is possible that certain methods used in the implementation are not the most efficient. Future development could investigate the optimisation of the system.

Appendix A

Project Sources

The source code for this application was too large to be printed on paper and so has not been included in the document. The attached CD does contain the source however and should be viewed to better understand the project.

Bibliography

[1] CROCODILE CLIPS, 2007. *Crocodile Clips: Common Questions.* Available from: `http://www.crocodile-clips.com/s4_3.htm` [Accessed 15 April 2007]

[2] HANSEN, J.P., 2004. *TKGate.* Available from: `http://www.tkgate.org` [Accessed 24 November 2006]

[3] WESTHAGEN, A. 2005. *Benefits of Open Source.* Available from: `depts.washington.edu/rfpk/pdfs/openSource.pdf` [Accessed 29 November 2006]

[4] LOGISIM. 2006. *Logisim.* Available from: `http://ozark.hendrix.edu/~burch/logisim/index.html` [Accessed 24 November 2006]

[5] NATIONAL INSTRUMENTS. 2006. *SPICE Simulation Fundamentals.* Available from: `http://zone.ni.com/devzone/cda/tut/p/id/5413` [Accessed 24 November 2006]

[6] CROCODILE CLIPS. 2006. *Crocodile Clips.* Available from: `http://www.crocodile-clips.com` [Accessed 24 November 2006]

[7] VARIOUS AUTHORS, 2006. *Electrical Circuits.* Available from: `http://en.wikipedia.org/wiki/Electrical_circuit` [Accessed 12 November 2006].

[8] ENCYCLOPEDIA BRITANNICA, 2006. Available from: `http://www.britannica.com/eb/article-34365` [Accessed 18 November 2006]

[9] BROPHY, J.J., 1977. *Basic Electronics For Scientists.* 3rd ed. McGraw-Hill

[10] MCCALLA, W.J., 1988. *Fundamentals of Computer-Aided Circuit Simulation.* 1st ed. Kluwer Academic Publishers.

[11] PILLAGE, L.T., 1995. *Electronic Circuit and System Simulation Methods.* 1st ed. McGraw-Hill.

[12] ECIRCUIT CENTER, 2006. *SPICE Algorithm Overview.* Available from: `http://www.ecircuitcenter.com/SPICEtopics.htm` [Accessed 10 November 2006]

[13] ANTON, H. 2002. *Calculus.* 7th ed. Anton Books.

[14] CONNEL, M. 2004. *Programming Language Benchmarks.* Available from: `http://furryland.org/~mikec/bench/` [Accessed 1 December 2006]

[15] PYTHON SOFTWARE FOUNDATION. 2007. *Python Programming Language* Available from: `http://www.python.org/` [Accessed 12 April 2007]

[16] VARIOUS AUTHORS. 2007. *Wikipedia - Perl* Available from: `http://en.wikipedia.org/wiki/Perl/` [Accessed 12 April 2007]

[17] COLVIN, G. 1999. *Smart Pointers* Available from: `http://www.boost.org/libs/smart_ptr/smart_ptr.htm` [Accessed 13 April 2007]

[18] GNOME FOUNDATION. 2007. *Make Your Application Consistent* Available from: `http://developer.gnome.org/projects/gup/hig/2.0/principles-consistency.html` [Accessed 18 April 2007]