Extensible Hierarchical Object-Oriented Logic Simulation with an Adaptable Graphical User Interface

by

Donald C. Craig

A thesis submitted to the

School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

1996

St. John's Newfoundland

Abstract

Simulators provide an economical means of understanding and evaluating the performance of both abstract and real world systems. In addition to being efficient and easy to use, modern day simulators must be able to cope with the demands of increasing complexity within systems. Simulators must also be easily extensible so that the behaviour and performance of a wide variety of systems may be studied.

This report will outline the design and implementation of a utility which integrates an interactive, graphical design tool with a discrete event simulation engine that employs the object-oriented paradigm as a means of combating complexity. Central to the simulation technique is the concept of *local time*, in which each entity being simulated maintains its own notion of time throughout the simulation. This concept promotes component encapsulation and self-containment thereby facilitating the implementation of distributed event-driven simulators. Although the simulation domain described in this report will consist primarily of digital logic circuits, the simulation techniques should also be amenable to the simulation of any discrete event system.

The graphical user interface front-end to the simulator engine is designed to be easy to use, hence making the underlying simulator engine accessible to a wide audience. The implementation of the interface is loosely integrated with the simulator engine, thereby providing a high degree flexibility between the interface and the simulator itself. The interface and the simulator can each operate as distinct, self-contained applications. As a result, the simulator engine could be configured to employ a different graphical interface and the graphical interface can be adapted for a variety of existing text-based simulator engines.

Acknowledgements

Once again, I would like to thank my supervisor, Dr. Paul Gillard, for his patience and support over the past three years and for continuing as my supervisor during his sabbatical. His enlightening discussions and flexible style made my graduate program both educational and enjoyable.

I am indebted to the *Natural Sciences and Engineering Research Council* (NSERC) for their generous financial support during the first two years of my graduate program.

I would also like to thank both Mrs. Jane Foltz and Dr. Wlodek Zuberek for formally approving my admission to Graduate School. In addition, I'd like to thank Dr. Zuberek for gently (but firmly) reminding me of my academic responsibilities after I had resumed full-time employment outside the University prior to the completion of my thesis. His encouragement contributed to the timely completion of this report.

Thanks also go to several other members of the Department of Computer Science, including Dr. Rodrigue Byrne who made several suggestions for improving an earlier version of the implementation and Miss Elaine Boone, whose proofreading skills helped eliminate several typographical and grammatical errors in an earlier draft of this thesis. Of course, I take full responsibility for any mistakes or oversights remaining in this report.

Finally, I would like to express my gratitude to the thousands of individuals responsible for the development and distribution of robust and inexpensive software tools. Freely available operating systems and software packages such as *Linux*, *XFree86*, Tcl/Tk, LaTeX, the GNU C++ compiler and related tools contributed significantly to the successful completion of this project.

Contents

List of Tables

List of Figures

To my parents

Chapter 1

Introduction

Advances in technology invariably lead to the construction of systems with additional layers of complexity being wrapped around more primitive but equally complex sub-systems. In the future, these systems may then, in turn, become sub-systems of larger, even more complex, super-systems. Simulators provide a means by which such abstract and real world systems may be understood and evaluated by duplicating the behaviour of these systems through hardware and software. Formally, we can define simulation as:

> "...the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behaviour of the system of or evaluating various strategies (within the limits imposed by a criterion or a set of criteria) for the operation of the system." [26]

Simulators must adapt to increases in system complexity by permitting users to simulate a system at several conceptual levels. Unfortunately, the design and implementation of simulators is almost as complex as the systems being simulated. As a result, there has been a concerted effort by the software community to apply the latest advancements in software technology in an attempt to counteract this ever increasing complexity. Their efforts have lead to simulators which are easy to maintain and extend while at the same time preserving their relative efficiency. The prominent paradigm currently being used to implement simulators is the ubiquitous *object-oriented* [3] paradigm, in which software entities closely model their real world counterparts. This paradigm has been successfully employed to implement a wide variety of simulators ranging from sawmill production [22] to air base logistics [20].

This thesis represents a major enhancement to an existing discrete-event simulation engine [7] which employs distributed event queues as its primary mechanism for event management instead of the more commonly used global event queue. By adopting a concept of *local time*, it will be shown how the global event queue may be replaced by several distributed queues, each of which are self contained within a simulation component. The system being simulated can then be decomposed hierarchically into several such components thereby promoting extensibility and modularity of the simulation system. Details will also be provided regarding how this simulation technique addresses the need for extensibility and how the simulation entities may be reused from system to system so as to avoid needless duplication of effort when constructing a new system. The advantages and disadvantages of using a global queue versus distributed queues will be discussed as the two simulation techniques are compared from a design, implementation and philosophical perspective.

In addition to describing and comparing the two simulation strategies described above, this report will also discuss the design and implementation issues associated with providing the simulator engines with a graphical user interface. Such an interface can be used to layout and connect the simulation entities, observe the dynamics of the system during simulation and collect various reports upon completion of the simulation for verification or performance evaluation. The ability to observe the dynamic interaction between components makes the simulator an ideal tool for understanding subtle aspects of a particular design or to teach novice designers the fundamental behaviour of elementary systems.

Before presenting details about the design and implementation of a distributed queue simulator, this chapter will provide a general overview of simulation. The need for simulators will be explained as will the potential problems that may arise through their imprudent use. After discussing some different simulation models, a few practical uses of simulators will be described.

1.1 The Need for Simulation

A simulator is a collection of hardware and software systems which are used to mimic the behaviour of some entity or phenomenon. Typically, the entity or phenomenon being simulated is from the domain of the tangible — ranging from the operation of integrated circuits to behaviour of a light aircraft during wind sheer. Simulators may also be used to analyze and verify theoretical models which may be too difficult to grasp from a purely conceptual level. Such phenomenon range from examination of black holes to the study of highly abstract models of computation. As such, simulators provide a crucial role in both industry and academia.

Despite the increasing recognition of simulators as a viable and necessary research tool, one must constantly be aware of the potential problems which simulators may introduce. Many of the problems are related to the computational limitations of existing hardware platforms but are quickly being overcome as more powerful platforms are introduced. Other problems, unfortunately, are inherent within simulators and are related to the complexity associated with the systems being simulated. This section highlights some of the major advantages and disadvantages posed by modern day simulators.

1.1.1 Advantages of Simulation

One of the primary advantages of simulators is that they are able to provide users with practical feedback when designing real world systems. This allows the designer to determine the correctness and efficiency of a design before the system is actually constructed. Consequently, the user may explore the merits of alternative designs without actually physically building the systems. By investigating the effects of specific design decisions during the design phase rather than the construction phase, the overall cost of building the system diminishes significantly. As an example, consider the design and fabrication of integrated circuits. During the design phase, the designer is presented with a myriad of decisions regarding such things as the placement of components and the routing of the connecting wires. It would be very costly to actually fabricate all of the potential designs as a means of evaluating their respective performance. Through the use of a simulator, however, the user may investigate the relative superiority of each design without actually fabricating the circuits themselves. By mimicking the behaviour of the designs, the circuit simulator is able to provide the designer with information pertaining to the correctness and efficiency of alternate designs. After carefully weighing the ramifications of each design, the best circuit may then be fabricated.

Another benefit of simulators is that they permit system designers to study a problem at several different levels of abstraction. By approaching a system at a higher level of abstraction, the designer is better able to understand the behaviours and interactions of all the high level components within the system and is therefore better equipped to counteract the complexity of the overall system. This complexity may simply overwhelm the designer if the problem had been approached from a lower level. As the designer better understands the operation of the higher level components through the use of the simulator, the lower level components may then be designed and subsequently simulated for verification and performance evaluation. The entire system may be built based upon this "top-down" technique. This approach is often referred to as *hierarchical decomposition* [32] and is essential in any design tool and simulator which deals with the construction of complex systems. For example, with respect to circuits, it is often useful to think of a microprocessor in terms of its registers, arithmetic logic units, multiplexors and control units. A simulator which permits the construction, interconnection and subsequent simulation of these higher level entities is much more useful than a simulator which only lets the designer build and connect simple logic gates. Working at a higher level abstraction also facilitates *rapid prototyping* in which preliminary systems are designed quickly for the purpose of studying the feasibility and practicality of the high-level design.

Thirdly, simulators can be used as an effective means for teaching or demonstrating concepts to students. This is particularly true of simulators that make intelligent use of computer graphics and animation. Such simulators dynamically show the behaviour and relationship of all the simulated system's components, thereby providing the user with a meaningful understanding of the system's nature. Consider again, for example, a circuit simulator. By showing the paths taken by signals as inputs are consumed by components and outputs are produced over their respective fanout, the student can actually see what is happening within the circuit and is therefore left with a better understanding for the dynamics of the circuit. Such a simulator should also permit students to speed up, slow down, stop or even reverse a simulation as a means of aiding understanding. This is particularly true when simulating circuits which contain feedback loops or other operations which are not immediately intuitive upon an initial investigation.

During the presentation of the design and implementation of the simulator in this report, it will be shown how the above positive attributes have been or can be incorporated both in the simulator engine and its user interface.

1.1.2 Disadvantages of Simulation

Despite the advantages of simulation presented above, simulators, like most tools, do have their drawbacks. Many of these problems can be attributed to the computationally intensive processing required by some simulators. As a consequence, the results of the simulation may not be readily available after the simulation has started — an event that may occur instantaneously in the real world may actually take hours to mimic in a simulated environment. The delays may be due to an exceedingly large number of entities being simulated or due to the complex interactions that occur between the entities within the system being simulated. Consequently, these simulators are restricted by limited hardware platforms which cannot meet the computational demands of the simulator. However, as more powerful platforms and improved simulation techniques become available, this problem is becoming less of a concern.

One of the ways of combating the aforementioned complexity is to introduce simplifying assumptions or heuristics into the simulator engine. While this technique can dramatically reduce the simulation time, it may also give its users a false sense of security regarding the accuracy of the simulation results. For example, consider a circuit simulator which makes the simplifying assumption that a current passing through one wire does not adversely affect current flowing in an adjacent wire. Such an assumption may indeed reduce the time required for the circuit simulator to generate results. However, if the user places two wires of a circuit too close together during the design, the circuit, when fabricated may fail to operate correctly due to electromagnetic interference between the two wires. Even though the simulation may have shown no anomalies in a design, the circuit may still have flaws.

Another means of dealing with the computational complexity is to employ the hierarchical approach to design and simulation so as to permit the designer to operate at a higher level of design. However, this technique may introduce its own problems as well. By operating at too high an abstraction level, the designer may tend to oversimplify or even omit some of the lower level details of the system. If the level of abstraction is too high, then it may be impossible to actually build the device physically due to the lack of sufficiently detailed information within the design. Actual construction of the system will not be able to occur until the user provides low level information concerning the system's subcomponents. With respect to circuit design and fabrication, work is currently on going in the field of *silicon compilers* [10] which are able to convert high level designs of circuits and translate them accurately and efficiently into low level designs suitable for fabrication.

1.2 Classification of Simulation Systems

It is useful to classify the system being simulated into two separate categories depending upon the degree of randomness associated with the behaviour of the system in its simulated environment. For example, consider a simulated system consisting of a series of bank tellers who must provide transaction services to incoming customers. The length of time required for a teller to process a customer's transaction cannot usually be predetermined before the simulation is started. Consequently such a simulation system must introduce random behaviour to simulate the duration of each transaction. During the analysis of a real world banking system it may be discovered that the time required for a transaction occurs over some well known probability distribution. Hence the duration of each transaction may be generated from this distribution. A similar strategy may be adopted for the rate at which customers enter a bank. Through the introduction of this randomness, the results of a simulation may never be the same as a previous simulation. A system, such as this one, that relies heavily upon random behaviour is referred to as a *stochastic* system [23]. The results generated from a stochastic system are typically analyzed statistically in order to make conclusions regarding the behaviour of the system.

Conversely, a *deterministic* simulation system incorporates absolutely no random behaviour whatsoever. As such, the simulation results for a given set of inputs will always be identical. Simulations involving circuit behaviour are examples of deterministic systems. Supplying high signals to both inputs of a 2-input *NAND* gate will always produce a low signal on the gate's output, regardless of where the gate is located in the circuit's design hierarchy or when the inputs are received by the gate. In the context of circuit simulation, deterministic simulation is used to verify that a particular circuit design is behaving as expected — when the circuit is supplied with a given set of inputs, the circuit produces the expected outputs at the correct time. Although this report will focus primarily upon deterministic simulation systems, stochastic systems can also be simulated with modest modifications to the implementation.

1.3 Simulation Models

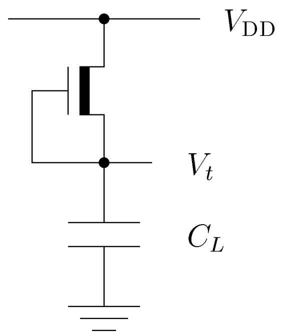
During the design and implementation of a simulator, various techniques and strategies may be adopted to model the behaviour of a given system. Depending upon the system to be simulated, some techniques may be more favourable than others. Factors including the level of abstraction and the desired accuracy and speed of the simulation should be taken into consideration when designing the simulator engine. Traditionally, simulators are designed using either *continuous* or *discrete-event* techniques to simulate a given system.

1.3.1 Continuous Simulation

Continuous simulators [4] are characterized by the extensive use of mathematical formulae which describe how a simulated component responds when subjected to various conditions. For example, consider a circuit described at the transistor, resistor and capacitor level. The behaviour of all these components are well understood and are governed by several equations which describe their respective behaviours. A continuous simulator would apply those equations in the context of the components' environment and connectivity and produce a continuous graph which accurately reflects how the components would react if they were actually hooked up in reality. The graphs usually reflect the changes in the state of the system with respect to time; however, other relationships may also be demonstrated as well. Unfortunately, the mathematical equations employed by a continuous simulator can make the simulation very computationally intensive, especially in the presence of thousands of interconnected elements. As such, continuous simulators may be slow and are consequently only useful when simulating a relatively small number of components which are described at a low level of abstraction.

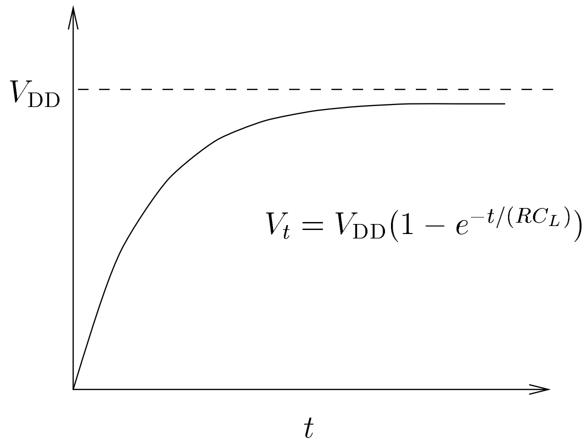
As an example of continuous simulation, consider a depletion mode transistor acting as a pull up for a capacitive load [17]. The schematic for such a device is presented in Figure 1.1. The transient behaviour of the system is governed by the equation $V_t = V_{\mathrm{DD}}(1 - e^{-t/(RC_L)})$.

Figure 1.1: Depletion Mode Transistor Pulling Up Capacitive Load



During the latter stages of the rising transient, a continuous simulator would produce the graph given by Figure 1.2.

Figure 1.2: Graph Representing Continuous Behaviour

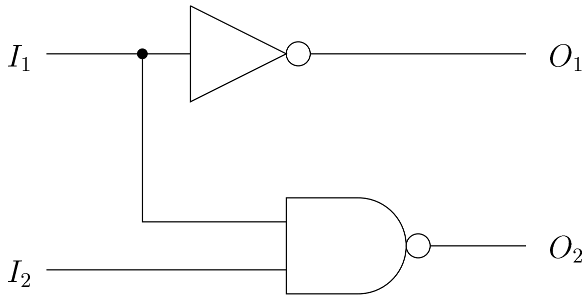


1.3.2 Discrete-Event Simulation

Discrete-event simulation [31] is used to simulate components which normally operate at a higher level of abstraction than components simulated by continuous simulators. Within the context of discrete-event simulation, an event is defined as an incident which causes the system to change its state in some way. For example, a new event is created whenever a simulation component generates output. A succession of these events provide an effective dynamic model of the system being simulated. What separates discrete-event simulation from continuous simulation is the fact that the events in a discrete-event simulator can occur only during a distinct unit of time during the simulation — events are not permitted to occur in between time units. Discrete event simulation is generally more popular than continuous simulation because it is usually faster while also providing a reasonably accurate approximation of a system's behaviour.

As an example of discrete-event simulation, consider the logic circuit presented in Figure 1.3.

Figure 1.3: Simple Digital Logic Circuit



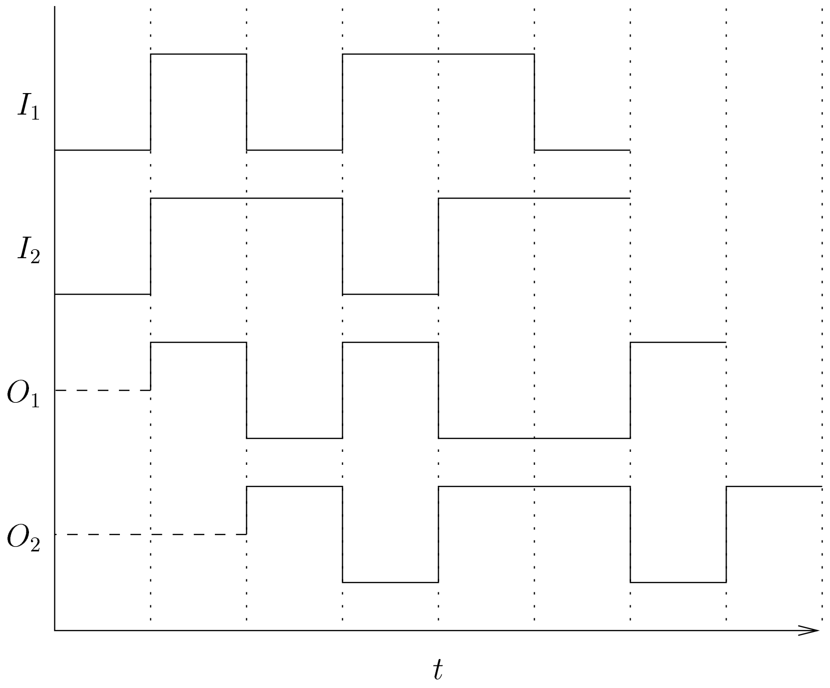
Assuming that the *NAND* gate has a delay of two time units and the *NOT* gate has a delay of one time unit, the above logic circuit will produce the outputs given in Figure 1.4 when supplied with the specified inputs.[1]

Conventionally, a data structure known as a *global event queue* is used to process and manage the events and to activate components as required during the simulation. This report will demonstrate an improved technique for event management in which the global queue is eliminated in favour of distributed event queues. The design and implementation of such a queuing system is discussed in detail later in this report.

Monte Carlo simulation is related to discrete-event simulation. Monte Carlo simulators usually make extensive use of random number generators in order to simulate the desired system. Unlike discrete-event simulators, which are often used to model

[1]For the purpose of this example, it is assumed that prior to the simulation, both gates are generating indeterminate outputs. These indeterminate outputs are represented in the diagram by the horizontal dashed lines.

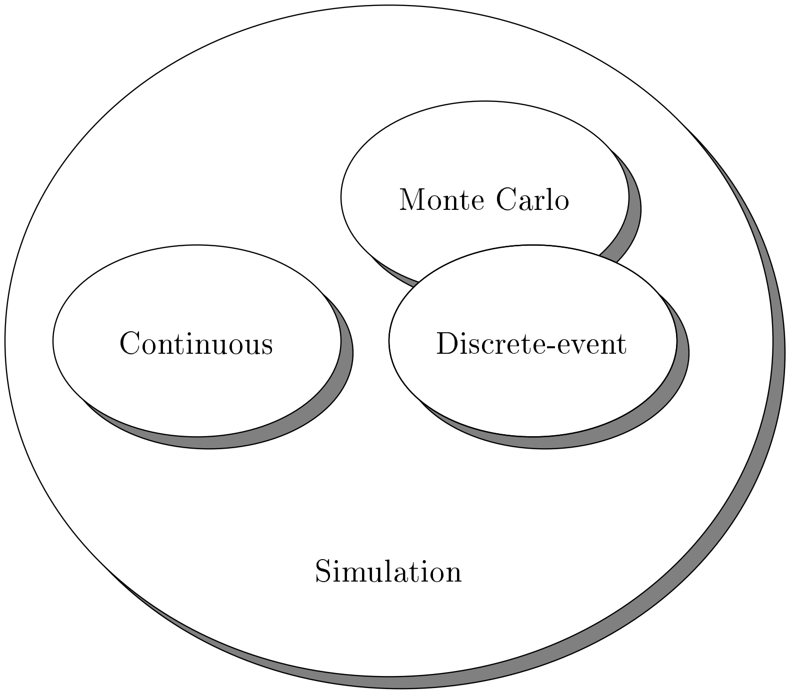
Figure 1.4: Graph Representing Discrete Behaviour



deterministic systems, Monte Carlo simulators can be used to effectively model systems in which probability and nondeterminism plays a major role. As such, Monte Carlo simulators are commonly used to model stochastic systems. The relationship between the three types of simulations is displayed in Figure 1.5.

Hierarchical simulation, although not a simulation type by itself, may be used in conjunction with continuous or discrete even simulators to simplify the simulation process. Hierarchical simulation is a process whereby higher order components delegate behavioural responsibility to its composite subcomponents. The higher level components are responsible for activating their respective child components in a meaningful sequence so as to model the correct behaviour of the system. As mentioned earlier, hierarchical design and simulation is one of the techniques used to cope with the complexity associated with a given system. Later chapters will demonstrate how the

Figure 1.5: Simulation Models



object-oriented paradigm lends itself very well to the description and simulation of hierarchical systems.

1.4 Abstraction Levels for Circuit Simulation

This report will describe the design and implementation of a digital circuit simulator. Consequently, this section will outline some of the aspects related to the simulation of circuits, including the different levels at which circuit simulators may operate.

Circuit simulation provides a means of modelling a circuit's response to a given set of inputs. The simulator may generate numbers representing the voltages present at specific nodes of the circuit at certain times or it may generate waveform diagrams that show the circuit's output over the duration of the simulation. What is actually produced as a result of the simulation is largely dependent upon the abstraction level at which the circuit was described and subsequently simulated. For example, simulating a circuit described in terms of transistors and capacitors will conventionally show how these components interact at the electrical or analog level, whereas simulating a circuit described in terms of gates, flip-flops and registers will demonstrate the digital interaction amongst the components.

Circuits may be described and simulated at several levels of abstraction [25]. This section will describe three major abstraction levels, each of which are related to the simulation models described above. These levels are described below in order of increasing abstraction level.

1.4.1 Circuit-level Simulators

Circuit-level simulators [24] are used to model the behaviour of a circuit at its lowest conceptual level. The circuit is described in terms of transistors, wires, capacitors and resistors and their respective interconnectivity. Circuit-level simulators manipulate extensive detail regarding the interaction of all the components in the circuit and also take into consideration subtleties such as wire resistance and geometric properties of the subcomponents. The end goal circuit level simulation is to produce very detailed analog waveforms which accurately model the behaviour of the circuit's devices in the real world. Consequently, continuous simulation techniques are often used to implement circuit-level simulators.

Circuit-level simulations are typically performed in several stages. During the first stage, referred to as *node-extraction*, static analysis of the circuit description is performed. From this analysis, information regarding the circuit's devices, their respective attributes and their connectivity is obtained. This information is subsequently combined with modules known as *device models* that describe the behaviour of each device on a mathematical level. In order to model the circuit's behaviour, the simulator must then solve a system of differential linear equations which is derived from all the information supplied to it during the node-extraction phase.

Although this method generates very accurate results, the technique is very computationally intensive, resulting in poor simulation speed. As a result, circuit-level simulation is usually not feasible for large designs and is therefore commonly used to simulate only the most critical subregions of a given circuit.

1.4.2 Logic-level Simulators

Logic-level simulators attempt to remedy the computationally intensive nature of circuit-level simulators by raising the level of abstraction to the domain of switches and logic components. Instead of manipulating continuous, analog data, logic-level simulators simply process logic values; that is, *0*, *1* and *X*. In addition, logic-level simulators traditionally simplify the simulation process by assuming that the connecting wires have negligible resistance.

Logic level simulators can be subdivided into two further categories, *switch-level* and *gate-level simulators*. In switch-level simulators, transistors are promoted to elementary switches and very little attention is given to the intricacies of other transistor attributes. During the simulation itself, equations governing the behaviour of the circuit are greatly approximated, thereby increasing the speed at which the simulator operates. The detail that is inevitably lost as a result of this approach is not usually vital.

Gate-level simulators [9] operate at yet a higher abstraction level. Low level circuit devices such as transistors, capacitors and resistors are replaced with logic gates such as *NAND*, *XOR* and flip-flops. Circuits described at this level bear strong resemblance to data flow diagrams in which information is passed amongst interconnected components. Effective use of logic gates permit relatively high-level designs to be easily described and subsequently simulated. Because the components are increasingly abstract, more complex systems may be designed and simulated at the gate level rather than at the switch level. Traditional gate-level simulators are implemented using discrete-event simulation; as such, this report will primarily focus upon circuits described at the gate level.

There have been several successful attempts to merge switch-level simulators with simulators that operate at the gate-level and above, thereby allowing the designer to have the flexibility and speed of high-level simulators, while at the same time retaining some of the accuracy associated with switch-level simulators. Such simulators are commonly referred to as *mixed-mode* simulators [8].

1.4.3 Functional- and Behavioural-level Simulators

Functional- and behavioural-level description languages and simulators [11] represent the highest levels of simulation available to circuit designers. These levels enable designers to model circuits in terms of interacting abstract units that may not even be capable of fabrication. As such, designers are not limited by the restricted behaviour of fundamental circuit devices. In addition, these levels also provide designers a viable means of quickly exploring alternatives without becoming overwhelmed with the impact that design decisions would have on the circuit at lower levels.

Functional-level simulators are generally closer to the actual hardware representation than behaviour-level simulators. An abstract unit in a functional-level simulation would accept input and produce output just like its corresponding hardware component. However, more flexibility is permitted with respect to how the input is presented to the unit and how it is processed to produce output. For example, an adder at the gate-level may consist of several half-adders which adds to numbers by operating directly on their bits. The equivalent functional-level unit would simply take two integers and add them using arithmetic constructs available in the hardware description language.

Behavioural-level simulators go one level higher and permit designers to model abstract control processing which may not be realizable in hardware. The purpose of these simulators is to give the designer a general overview of the design and to experiment with high-level alternatives. The usefulness of design tools and simulators that operate at this level has been the subject of debate due to the difficulty in translating such high level designs into compact, high-performance circuits. However, advances in silicon compilation have made the translation process easier and more efficient. In addition, high level simulators are still useful for rapid prototyping, even if the design is not actually physically fabricated. Rapid prototyping allow designers to study the feasibility of a high level design before actually delving into the tedious, low-level details such as placement and routing.

One of the more popular languages for describing, simulating and eventually synthesizing circuits at the functional and behavioural level is VHDL [6][1].

1.5 The Purpose of this Report

The simulation engine presented in this report provides extensible support for a variety of circuits each of which may be described at different levels of abstraction. The class structure is particularly amenable towards the specification of high-level functional blocks that can be easily described in C++ — the language used as the basis for hardware simulation by the implementation described herein. A user-defined library consisting of high- and low-level components can be designed and integrated easily with the core library components. Subsequent chapters discuss all aspects of the simulator in detail. These details include implementation concerns with respect to the graphical user interface, the simulator engine core as well as the means by which these two major components communicate with one another.

Chapter 2

The Simulator User Interface

The intuitiveness and robustness of the user interface used by a software application can strongly influence the productivity of the people using that application. This chapter focuses on the design and operation of a graphical user interface for a digital simulator engine. Details with respect to the chosen platform and implementation language will also be discussed and justified. In addition, some limitations of the GUI will be described and potential solutions to these restrictions will be presented.

2.1 Motivation

The core of the discrete-event simulator engine described in this paper was previously designed and implemented as part of an Honours project. Despite the capabilities of the simulator engine, the implementation was limited from an end user perspective. One of the biggest problems of the implementation was the user interface — or rather, the lack thereof. In order to describe a circuit, the user had to define the entire circuit as well as the input signals in a C++ source module. The circuit description, inputs and simulator engine itself then had to be compiled, linked and executed in order to determine the outputs from the circuit. The output from the simulator consisted of time stamps and signal values which were displayed textually rather than graphically. Consequently, verification of the behaviour of the circuit was often difficult and tedious. In addition, any changes to the circuit description or input signals, regardless of how small, required modification and subsequent recompilation of the source code. Needless to say, this technique for circuit specification and simulation seriously compromised both the evaluation and practicality of the simulator engine itself.

In order to get around these problems, a graphical user interface [19] (or *GUI*), called *DigiTcl*, was designed and implemented for the simulator engine. The benefits of such an interface are multifold. Firstly, due to the graphical representation of such an interface, further investigation into the feasibility, practicality and accuracy of the simulator engine becomes easier. Instead of verifying the reliability of the simulator by examining streams of textual output, waveforms can be generated and studied. Secondly, if the GUI is designed and implemented correctly, it can be used as the front end for a multitude of circuit simulators. This feature would make it significantly easier to compare the performance and capabilities of a variety of different simulator engines. In order to implement this feature, loose coupling [27] between the simulator engine and the GUI is necessary. Thirdly, a GUI would make the specification of circuits easier for novice end users, thereby making the simulator engine more accessible to people who may not be familiar with relatively esoteric concepts such as source code compilation. An intuitive GUI would facilitate circuit specification and design, hence increasing the academic and even industrial applications of the simulator engine.

2.2 GUI Platform and Implementation Language

Unfortunately, the choice of a viable platform and implementation language for any software project can often be considered a religious issue at best. In selecting a hardware platform and implementation language for the circuit editor GUI several criteria, such as cost, availability and level of support were taken into consideration. The list of possible platforms and languages discussed below is by no means exhaustive, but they do represent some of the more popular options available.

2.2.1 GUI Platform

Linux [29] was chosen as the operating system/platform upon which the circuit editor would be developed. *Linux* is a freely distributable clone of the *UNIX*[1] operating system which has been ported to a variety of hardware architectures. The operating system offers stability, open development, source code availability and a variety of suitable software development tools. Also, it runs on relatively cheap hardware. Other excellent *UNIX*-like operating systems, such as *FreeBSD* and *NetBSD*, possess similar qualifications and were also potential candidates, but it was decided to go with *Linux* due to familiarity with this system and accompanying distributions.

As a result of this decision, it was decided that the circuit editor GUI should run on the windowing system most prevalent on the *UNIX* platform, namely, the *X* Window System[2] [21]. As luck would have it, *XFree86*[3] is a freely available and robust port of the *X* Window System to PC based *UNIX* systems. Despite the decision to

[1] *UNIX* is a registered trademark of X/Open Company, Ltd.
[2] *X* Window System is a trademark of X Consortium, Inc.
[3] *XFree86* is a trademark of The XFree86 Project, Inc.

use *UNIX* and *X* for development, the chosen implementation language should be portable, thereby allowing the GUI to be used with other operating systems should the need arise.

2.2.2 Implementation Language

The choice of an implementation language is somewhat less clear cut. The most obvious contender would be C [14] in conjunction with the *Xlib* library. However, it was deemed that this library was too low-level for rapid application development. Another possibility was the use of C++ [28] and the *Fresco* class library. While it is higher level than *Xlib* and shows great promise, it still represents, at the time of writing, a work in progress; hence, its potential usability and stability is questionable. *InterViews* [15], the forerunner of *Fresco*, was another considered as another possible option; however, support for this class library appears to have been abandoned in favour of *Fresco*. Finally, a scripting language known as Tcl/Tk [2][18] was evaluated and eventually chosen as the desired implementation language.

Tcl (*Tool Command Language*) is a simple scripting language that is being developed by Sun Microsystems[4] which provides support for common programming concepts such as variables, control flow, procedures and string manipulation. Tcl scripts can be used either as a stand-alone language or they may also be embedded in C code, thereby interfacing with existing libraries and legacy code. Tk (*Toolkit*) is an extension of Tcl which can be used for implementing graphical user interfaces for the *X* Window System. Tk provides the programmer with a wide variety of widgets (such as buttons, listboxes, canvases and scrollbars) which can be configured and arranged

[4]Sun Microsystems is a trademark of Sun Microsystems, Inc.

in a flexible manner to build a robust GUI.

Advantages of Tcl/Tk

One of the primary advantages of Tcl/Tk is that the source code is freely available on the Internet from Sun Microsystems. Consequently, there is no need to deal with the economic burden nor the administrative overhead of paying for the package initially and paying again for subsequent upgrades and bug fixes. The author of the package, John Ousterhout, has adamantly stated that the Tcl core and Tk extension will always be freely available. In addition, neither licenses nor royalties are required when distributing applications built with the language.

By making the source code freely available, two other advantages arise. First, that fact that Tcl/Tk is free has undoubtedly contributed to its widespread use. The Tcl/Tk community is estimated to number in the tens of thousands, therefore providing the new user with a well established user-base to fall back on for assistance and guidance. This user-base is easily reached via the Usenet newsgroup `comp.lang.tcl`. Second, freely distributing the source code leads to open development of the package. End users are free to fix bugs and make suggestions and enhancements to the existing Tcl core. The existence of a clean, well-documented functional interface to the internal mechanisms of Tcl makes it relatively easy to extend Tcl to include features which are either too slow or not directly supported in Tcl. If the extensions are deemed useful to the Tcl community as a whole, then these extensions may be integrated into the core in the next release for the benefit of all users.

Programming a GUI can be a very arduous and demanding chore. Tcl/Tk helps make the task easier by raising the level of abstraction for the programmer, thereby making the implementation of user interfaces easier and quicker. Graphical interfaces written using Tcl/Tk typically require significantly less code than an equivalent interface written in C. Tcl is relatively easy to learn and provides most of the features one would expect from a general purpose programming language. Since Tcl is an interpreted scripting language, there is no need for the developer to compile the code. This makes rapid prototyping more feasible with Tcl/Tk.

Tcl/Tk was originally implemented for the *X* Window System; as a result, it runs seamlessly under a wide variety of *UNIX* platforms, including *Linux*. At the time of writing, ports were in progress to other popular operating systems, thereby enabling an application written using Tcl/Tk to be relatively portable across a variety of different architectures and operating systems. The potential user-base of such an application is, therefore, quite large.

Disadvantages of Tcl/Tk

Despite the numerous advantages of Tcl/Tk, there are also a few shortcomings of the language which must be taken into consideration when writing scripts. Some of these problems can be overcome by adopting disciplined programming practices, while others may be corrected by extension packages, many of which are also freely available. This section outlines many of the potential drawbacks of Tcl/Tk in the context of the circuit editor GUI implementation.

Because Tcl scripts are interpreted instead of compiled, execution of the scripts will obviously be slower than an equivalent C or C++ implementation. However, as the implementation of the circuit editor progressed, it was discovered there were only two situations in which speed played a major factor — the extraction of netlists and movement of multiple circuit elements. To alleviate the former problem, a feature known as *dynamic netlist identification* was implemented and is discussed in Section 3.6.1. The slowness resulting from the latter problem could have been corrected by moving only a rectangular outline enclosing the circuit elements being moved, but this feature was not yet implemented at the time of writing. Of course, the option to rewrite these slow operations in C and build a new Tcl interpreter is always possible should the need arise.

Another potential problem has to do with the fact that the simulator engine was already written in C++ and not Tcl. By adopting Tcl/Tk as the language for the GUI, we must establish some means of communication between the two different implementation languages so that the GUI can inform the simulator what circuit to simulate and the simulator could report the simulation results back to the GUI for presentation to the end user. Although there are ways of embedding Tcl/Tk within a C++ application, the decision was made to keep the GUI and the simulator modules distinct from one another and instead to link them together via a bidirectional pipe. Communication between the two modules would then take place using a well defined protocol. In addition to solving the inter-module communication problem, this physical separation of abstractions encourages (indeed, enforces) loose coupling between the GUI and the simulator engine, thereby resulting in a more flexible and orthogonal implementation. The details of this protocol are described in Chapter 5.

One of the more serious shortcomings of Tcl/Tk is the lack of a rich set of data structures. The only true data types in Tcl are *strings* and *associative arrays*.[5] The *string* type is also used to provide support for integers and floating point numbers and

[5]Some purists argue that *associative arrays* are not true types in Tcl.

for lists as well. The lack of aggregate data structures increases the need for global variables as a means of inter-procedural and inter-module communication in situations where procedural parameters are not possible. This problem is compounded by the lack of namespaces which can seriously compromise an effective modular design. As a result, Tcl scripts do not scale very well. However, there are several possible workarounds to these inherent problems. In order to compensate for the lack of namespaces, a prefixing scheme was devised for procedures and global variables which would reduce the potential for clashes between variable and procedure names across different modules. Also, several extension packages for Tcl exist which offer better support for data abstraction and enhanced scoping. Alternatively, algorithms involving a rich set of data structures could be written in C and linked with the Tcl core library.

As mentioned earlier, several freely available extensions to Tcl/Tk exist which help to overcome many of the above problems. However, the decision was made not to use them due to the relative volatility of Tcl/Tk during the implementation of the circuit editor GUI. During this time, Tk was undergoing a major revision from 3.6 to 4.0, which rendered some of the extension packages unusable due to the many backward incompatible changes introduced into the new version of Tk. While some of the extension packages bravely kept pace with the numerous changes to the Tcl/Tk core, other extension packages have adapted more slowly. However, as these packages are upgraded to adopt the new features of Tk 4.0 and become more mature, an effort may be made to reevaluate and possibly reimplement the GUI using one of these extension languages.

2.3 Overview of the Simulator GUI

At the beginning of this chapter, the importance of the GUI was emphasized from the perspective of both the novice user learning about digital circuits to the experienced user who may wish to explore different event-driven simulation techniques. The GUI must therefore be easy to use for both new users and experts alike.

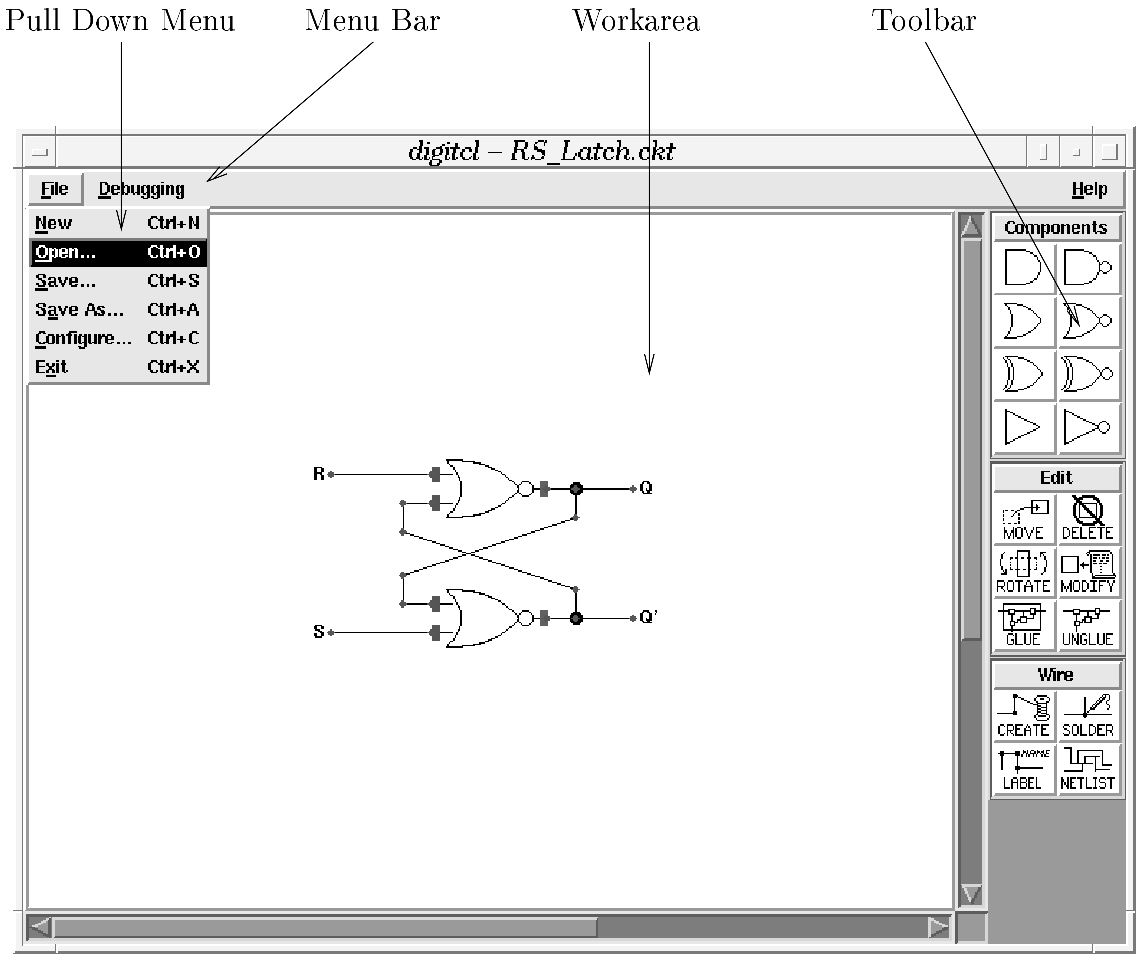
This section provides a high level description of the GUI as seen by the end user; it constitutes an abbreviated user's manual, which describes how the user interacts with the GUI to layout logic circuits, specify the circuit inputs and generate simulation output. Some lower level details are also presented to describe the implementation of some of the top level interface elements. More implementation details, especially with respect to internal representations are presented in Chapter 3.

The simulator GUI employs two windows — the *circuit editor window* (also known as the *main* window) and the *signal display window*. Together, these two windows provide the necessary functionality which lets the user construct and simulate digital circuits. Both of these windows are discussed in further detail in the subsequent subsections.

2.3.1 Circuit Editor Window

The circuit editor window serves as the main window of the entire application. Using the features provided by this window, the user can create, modify, save and load gate-level circuit diagrams. In order to make the circuit editor easy to use, the main window adopts a presentation format which has been adopted by numerous other GUI applications — it employs a pull-down menu bar, toolbar and workarea arranged as shown in Figure 2.1. By using a GUI layout which is already prevalent in industry, users who have experience with a similar interface layout should find the circuit editor relatively easy to use.

Figure 2.1: Circuit Editor Window



The following subsections briefly describe the user interface elements which comprise the main window display. In particular, an overview will be provided regarding their purpose, usage and implementation.

Pull-Down Menu

The *pull-down menu* user interface component is ubiquitous in software applications today. It is comprised of a row of menu buttons, called a menu bar, along the top of the display each of which is associated with a pull-down menu. The pull-down menu (or submenu) is displayed when the user clicks the leftmost mouse button on the menu button. The user can then drag the mouse pointer to the desired option in the pull-down menu and release the mouse button to activate the feature.

Pull-down menus offer users both familiarity and ease of use. As users become acquainted with the features offered by the pull-down menus, accelerator keys may be used instead of the mouse to activate submenu options. The accelerator key for a given option is indicated to the right of the option label in the pull-down menu. In addition, the pull-down menu may be activated using the keyboard by pressing the **Alt** key (or equivalent) and the underlined letter of the menu button in the menu bar. The left and right arrow keys will traverse adjacent submenus while the up and down arrows can be used to select options within the submenu. The current implementation of the circuit editor offers three pull-down menus: File, Debugging and Help. By convention, the Help menu button is displayed to the right of the other menu buttons.

The File pull-down menu consists of six options, most of which are related to the loading and saving of circuit descriptions. The New option lets the user create a new circuit by erasing any circuit currently being edited on the display. The Open... option displays a dialog box which lets the user restore a circuit that had been previously saved. The Save... feature will save the circuit in a file previously selected by the user. The Configure... option displays a dialog box which lets the user modify certain aspects regarding the look and behaviour of the GUI. The configuration dialog box is discussed in detail in Section 2.4. Finally, the Exit option terminates the application.

The Debugging pull-down menu was used primarily during development and enables the user to obtain details regarding the internal configuration of the circuit currently being constructed. Information regarding the connectivity and the circuit description itself can be obtained using options present in this menu. Also, this menu can be used to display bounding boxes of various circuit elements in the workarea which is useful for verifying that overlap detection is working correctly.

The Help pull-down menu presently contains only one option, namely, About... which displays a dialog box containing the name and version of the application. In the future, this menu should offer full hypertext help to the user.

The options offered by the pull-down menu are sparse at present; however, new options can be made available to the user relatively easily through the use of a pull-down menu module written in Tcl/Tk. Details describing some of the implementation details as well as how to add new options is presented in Chapter 3.

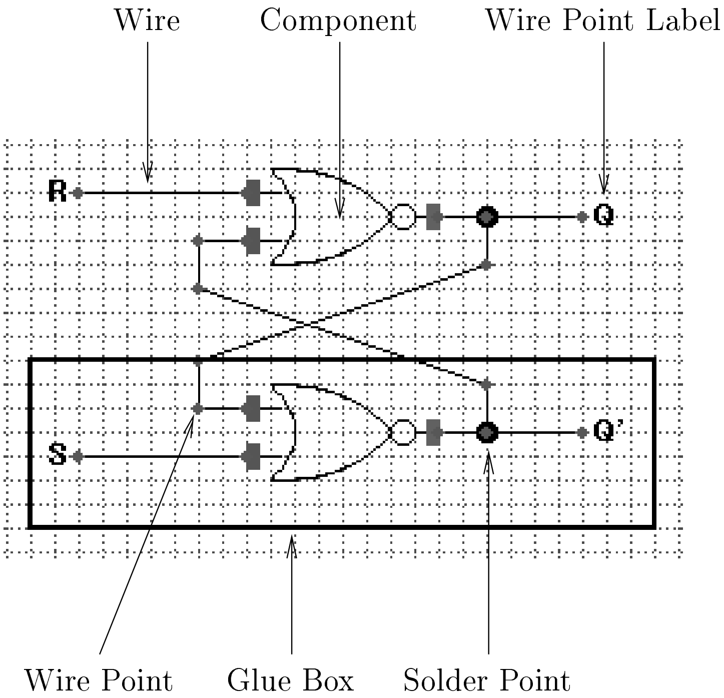
Workarea

The most dominant part of the circuit editor window is the *workarea* which is located under the pull-down menu. The workarea is the region where the interactive layout and placement of various circuit components and elements takes place. The workarea currently provides support for the placement of components, wires, wire points,[6]

[6] Wire points serve primarily as *handles* which make manipulation of the end points of wires easier for the end user.

solder points, wire point labels and glue boxes. Examples of each of these elements are provided in Figure 2.2. All these elements are created and modified with the help of the toolbar, which is described later. The workarea also displays a grid which can be used by the user as an aid for aligning gates and wires. The grid lines may be turned off using the configuration dialog box mentioned in Section 2.4. Along the right and bottom of the workarea are two scrollbars which can be used to scroll the workarea region vertically or horizontally in case the circuit design is too large to be viewed all at once.

Figure 2.2: Circuit Elements



The workarea is implemented using the Tk *canvas* widget which is used to display 2-D structured graphics. Primitives such as lines, ovals, and text can be created and configured in several versatile ways. Like other widgets, keyboard and mouse bindings may be applied to the canvas as a whole or to items on the canvas itself. This feature of Tk can be exploited to provide more feedback for the user, thereby making the user interface easier to use. For example, when a toolbar operation is selected, the bindings on the workarea canvas are changed so that whenever the mouse pointer enters a circuit element that will be affected by the operation, the circuit element will be displayed in a different colour. This makes it easier for the user to identify the target circuit element. If the user, for instance, presses the SOLDER toolbar button, and moves the mouse pointer over the port of a component, the port will be highlighted indicating that a soldering attempt will be made when the user presses the leftmost mouse button. If the user moves the mouse pointer over a component body, however, the component body will not change colour.

During the implementation of the GUI, an attempt was made to make mouse and keyboard actions on the workarea consistent. To this end, whenever the user presses the leftmost mouse button on the workarea, the operation selected from the toolbar would be performed; pressing the rightmost mouse button or the **Esc** key would terminate the operation. The actions performed whenever the middle mouse button is pressed depends upon the current context and the operation selected.

Toolbar

The pull-down menu described earlier is mainly used for administrative tasks, such as loading and saving circuit descriptions and GUI configuration settings. The actual creation of circuit elements is carried out with the help of the *toolbar*, from which the user selects logic gates to place on the workarea and operations to modify the circuit. Unlike the pull-down menu, the toolbar is accessible using the mouse only and not the keyboard.

The toolbar is comprised of a vertical column of iconic buttons along the side of the main window. The user may configure the GUI so as to have the toolbar displayed on either the left or right side of the main window. The toolbar provides three major operations, hence the partitioning of the toolbar into three main blocks.

The first block of operations relates to Component creation. Using these buttons, the user can quickly create and place logic gates on the workarea. Currently, the software supports the creation of *AND*, *NAND*, *OR*, *NOR*, *XOR*, *XNOR*, *BUFFER* and *NOT* gates. These gates are hardcoded in the current implementation of the script. However, future versions may consult a library upon initialization, during which the circuit editor will be informed of all the gates and components supported by the underlying simulator engine. These gates and components can then be presented to the user by the GUI.

After single-clicking the appropriate toolbar button with the mouse, the user moves the mouse pointer onto the canvas; the new logic gate will be created and will follow the mouse as it is dragged across the canvas. The user can place the component by clicking the leftmost mouse button at the desired location on the workarea. The gate may also be rotated before placement by clicking the middle mouse button; hitting the rightmost mouse button cancels the placement and removes the gate from the canvas. More details on the implementation of component creation is presented in Section 3.5.1.

The second block of buttons contains the Edit operations. Using these features, the user can move and delete circuit elements, rotate components, and modify various attributes of the circuit elements. These operations all behave in a similar manner, with the leftmost mouse button carrying out the operation on the selected circuit element and the rightmost mouse button terminating the operation. In the case of component rotation, the leftmost and middle mouse buttons rotate the target component 90° counter-clockwise and 90° clockwise, respectively. Any wires connected to the component are moved accordingly. This is accomplished by subjecting each of the wire points connected to the ports of the component to the same rotation matrix used to rotate the component itself.

The GLUE button on the Edit section lets the user glue several circuit items together by holding down the leftmost mouse button on the canvas and stroking out a bounding box with the mouse. All circuit elements which lie inside the box after the mouse button is released will be treated as a single entity. This makes it easier to perform an operation on several circuit items simultaneously. For example, several gates and wires could be glued together and then subsequently moved as a single unit. This makes manipulation of the circuit much easier than moving each of the elements individually. The unglue operation will remove the bounding box, thus enabling the user to once again manipulate the circuit elements as separate units.

The last block of buttons in the toolbar contains the Wire creation and manipulation operations. Using these buttons, the user can create, identify, label and solder netlists, which are responsible for connecting components and propagating signals.

The CREATE button changes the mouse pointer to a wire spool which is used to layout netlists on the workarea. The leftmost mouse button is used to lay down each of the wire points and the middle mouse button completes the wire. The rightmost mouse button aborts the netlist creation operation and removes all the points just created. During wire creation, rubber-banding of the wires is used so as to provide the user with immediate feedback of how the current wire will be placed. The user may configure the interface so that only vertical and horizontal (manhattan style) wires may be drawn. In addition, if the user sets a wire point on top of an existing wire, an existing wire point or a port, the new point will automatically be soldered to that item. These features may be enabled or disabled using the configuration dialog box described in Section 2.4.

The SOLDER button lets the user solder two netlists together or solder a netlist to the port of a component. After pressing this button, the mouse pointer will change into a soldering iron on the workarea. Clicking the leftmost mouse button on top of wires, wire points and ports will solder all the overlapping items together. All the items under the cursor will be soldered intelligently — the soldering feature prevents the creation of cycles in a netlist and prohibits the soldering of two output netlists together.[7] The soldering feature can also be used to introduce wire points along an existing wire. One important feature currently lacking in the circuit editor is the ability to "unsolder" wires. However, the same end result can be achieved by introducing a new wire point next to the soldering point and then deleting the intervening wire segment.

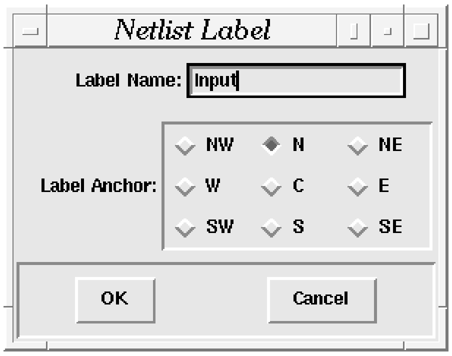
Once netlists have been created and soldered together, the NETLIST button can be used to identify netlists. After clicking this button and moving the mouse pointer to a netlist element (that is, a wire or wire point) on the workarea, all wires and points which comprise the corresponding netlist will be highlighted. This feature can be used to verify the connectivity of the circuit since the inter-connections of the circuit may

[7] Should the need arise to support tristate circuit elements, these soldering restrictions may be lifted by making minor modifications to the script.

not be immediately obvious based upon visual inspection alone. Note that netlist extraction is not done each time the button is pressed. Instead, we rely upon the dynamic netlist identification feature to identify and tag netlist items as wires and points are laid down and soldered. This is described more fully in Section 3.6.1.

Finally, the LABEL button can be used to name the netlists for both documentation and simulation purposes. After pressing this toolbar button, the user moves the mouse pointer to a wire point in the netlist that is to be named and clicks the leftmost mouse button. A dialog window, as shown in Figure 2.3, is displayed which lets the user supply a netlist name and specify the orientation of the label with respect to the point. After providing a name and orientation, the user presses the OK button, and a waveform corresponding to the label is created in the signal display window, which is described in Section 2.3.2.

Figure 2.3: Netlist Label Dialog Box



Note that only wire points and *not* the wires themselves can be labelled. Also, the interface prohibits the user giving two netlists the same name or a single netlist two names. As a consequence, the user cannot solder together two netlists which have already been labelled. The name and orientation of labels may be changed using the

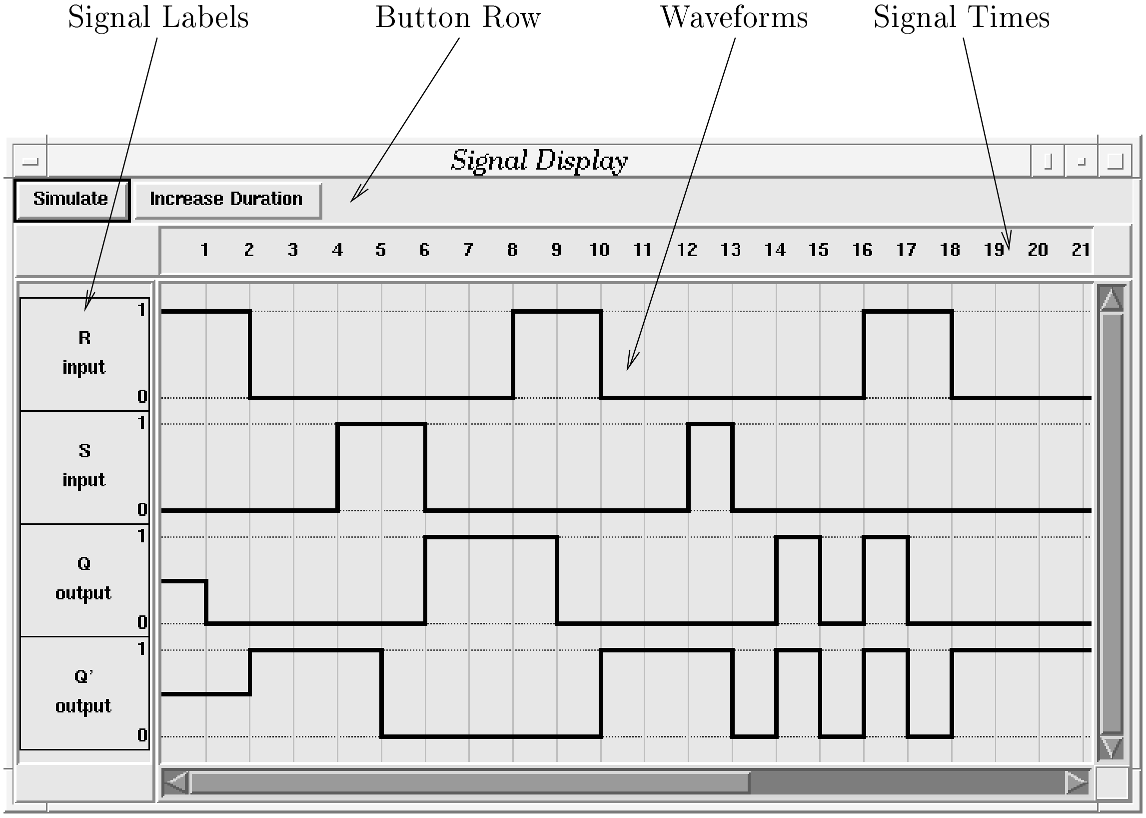
MODIFY button of the Edit section of the toolbar. Labels may also be deleted using the DELETE button of the toolbar. After the deletion of the label, the corresponding waveform in the signal display window will also be removed.

2.3.2 Signal Display Window

The second top-level window displayed by the GUI upon startup is the signal display window, presented in Figure 2.4. The purpose of the signal display window is to let the user edit the signals for each of the labelled input netlists and view the corresponding output signals generated from the circuit simulation. The signal display window is broken down into four major sections consisting of the signal waveforms, button bar, signal times and signal labels.

The largest portion of the signal display window is devoted to the *signal waveform* area. This section of the window displays the input/output waveforms which represent the signals that have travelled along each of the labelled netlists. In addition to displaying the input and output signals, this section of the window also lets the user modify the input signals to be processed by the circuit. To modify a signal, the user moves the mouse pointer to the waveform to be modified and then uses the leftmost mouse button to pull signals in the waveform high or low. The user may also "draw" the input waveform by clicking and holding the leftmost mouse button and dragging the mouse across the waveform. The signal values for the waveform will snap to the discrete signal value closest to the mouse pointer as the mouse is dragged. Currently, the waveform editor supports three discrete values: *high*, *low* and *unknown*; the latter of which is represented by a horizontal line in the middle

Figure 2.4: Signal Display Window



of the high and low boundaries. The GUI explicitly prevents the user from directly modifying waveforms which are not input signals.

The *button bar* at the top of the window currently contains only two buttons, but it is relatively easy to add extra buttons should the need arise for extra functionality. These buttons provide access to features which directly affect the signal display window and are therefore located here instead of in the circuit editor window. When the user presses the Simulate button, the application first ensures that the circuit is complete (that is, the ports of all the components have been connected to a netlist). After this check, descriptions of the circuit and its input signals are composed and sent to the simulator engine for simulation. After simulation, the results are fed back to the GUI which then parses the results and displays the subsequent output waveforms in the signal display window. The Increase Duration button simply adds ten time units to the time line so that signals in the waveform area can be extended further in time. Unfortunately, the total duration of the signals cannot yet be decreased.

The *signal times* portion lies immediately beneath the button bar and runs horizontally, thereby annotating the waveforms beneath it. This portion of the signal display window is non-interactive and serves only to denote the time of the signal values beneath it.

The *signal labels* section runs vertically down the left hand side of the window and is divided into subsections, each of which represents a labelled netlist in the circuit editor window. Each subsection contains the name and type of the netlist it represents and annotates the signal waveform immediately to its right. Signals may be rearranged by clicking and holding the leftmost mouse button on the signal label to be moved and then vertically dragging the signal label to the desired location. After the mouse button is released, both the signal label and it corresponding waveform will be moved to the new location and all other signal labels and waveforms will be adjusted accordingly.

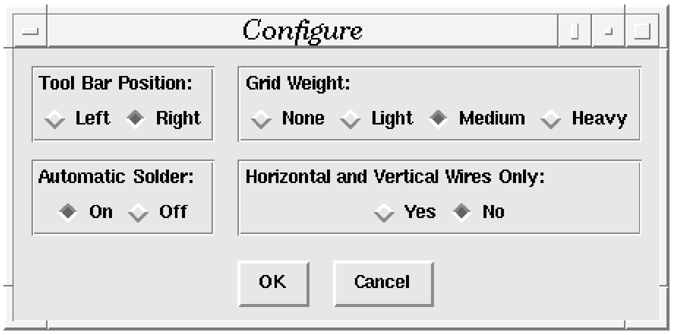
In the current implementation, netlists have three types, *input*, *output* and *unknown*; the type of each netlist is displayed below its name in the signal labels section. By default, all unconnected netlists created by the user are given the type *unknown*. As netlists are soldered to the ports of components, the type of the netlist changes automatically in the signal labels section to reflect their new type status. For example, if the user connects an isolated netlist to the input port of a component, then the word unknown will be replaced by the word input below the label name. The complete rules for netlist type determination are outlined in Section 3.6.2.

Like the workarea of the circuit editor, the signal display also employs two scrollbars. However, in addition to scrolling the waveforms, these scrollbars also scroll the signal labels or signal times as the user scrolls the vertical or horizontal scrollbar respectively. Hence the signal labels and times always line up correctly with the waveforms.

2.4 Configuration Options and Resource Database

GUI designers cannot always anticipate the needs and preferences of potential end users. As a result, it is common for user interfaces to be customizable in their look and behaviour so as to fit the needs of the widest user base possible. The circuit editor GUI can be customized via the configuration dialog box as shown in Figure 2.5. This dialog box is displayed when the user selects the Configure... option from the File menu of the menu bar.

Figure 2.5: Configure Dialog Box



Currently, only four aspects of the GUI can be configured using this dialog box. These configuration parameters are selected by pressing the appropriate radio-buttons in the dialog box.

Tool Bar Position This section of the configuration dialog box lets the user change the location of the toolbar to either the left or right side of the circuit editor window. Pressing either radio-button moves the toolbar immediately, so that the user can evaluate the decision without having to first leave the dialog box. By default, the toolbar is displayed on the left side of the circuit editor window.

Grid Weight The intensity of the grid which is displayed in the workarea canvas may be modified to one of three settings so as to increase or decrease the visibility of the grid lines. The grid lines may also be turned off completely. Once the user selects the appropriate radio button, the grid intensity is changed immediately on the workarea, thereby giving the user instant feedback.

Automatic Solder During netlist creation, when the user sets a point on top of an existing wire, wire point or component port, an attempt will be made to solder the new point to the overlapping item. If this behaviour is unacceptable to the user, then this feature can be turned off.

Horizontal and Vertical Wires Only The user has the option of configuring the GUI to prohibit the drawing of slanted lines during netlist creation. If the user is doing a manhattan style layout, then this restriction would make such a layout easier. Note that this only affects netlist creation — the user may subsequently move wires and wire points which would result in oblique wires.

After the user presses the OK button, a file named `.digitclrc` is automatically updated with the new configuration information in the user's home directory. Hence, the next time that the user starts the application, all the previous settings will be restored. If the user selects the Cancel button, the configuration settings will revert back to what they were before the configuration dialog box was activated.

Other aspects of the GUI can be configured using the X *option database*. Using this technique, virtually everything from the colour of circuit elements in the workarea of the circuit editor to the widths of the signals displayed in the signal display window can be customized by the user. Unfortunately, it is not easy to use the option database to configure the GUI. One must be familiar with the widget hierarchy employed by the application as well as be aware of the valid settings each configuration option can take. All the configurable aspects of the GUI are stored in the file `optiondb.tcl` which contains a procedure that assigns default values to option names. Any modifications made to the option values will take effect the next time the application is run. Alternatively, one can modify an option setting through the use of an X resource database such as the `.Xdefaults` file. For example, if the user wanted to change the colour used to display a wire when it is highlighted, the following line can be added to the user's `.Xdefaults` file:

```
digitcl.workarea.canvas.wireColourSelected:  green
```

and the option database would have to be re-read by issuing a command such as:

```
$ xrdb -merge ~/.Xdefaults
```

When the GUI is restarted, wires will be coloured green as the mouse pointer moves over them during, for example, a delete operation.

2.5 GUI Limitations

Despite the potential usefulness and intuitiveness of the interface, some major shortcomings and several minor deficiencies were identified in the current implementation of the GUI. While many of the problems are relatively simple to correct, others may require non-trivial modifications to the implementation. This section discusses some of the major limitations.

Among the limitations, perhaps the most obvious is the current inability to represent circuit designs hierarchically — circuit layout can take place at one level of abstraction only. The addition of hierarchical decomposition would make the circuit editor more pragmatic in both academic and industrial settings as this would permit the construction of custom libraries containing components which could be easily reused in future designs. Hierarchical representation would require the ability for the user to stroke out and encapsulate components in a block, in much the same fashion as the glue feature operates. The user would then have the opportunity to save that portion of the circuit in a library of so-called *mega-components*. During subsequent designs, the user would have the ability to integrate these mega-components into future designs instead of having to recreate them from scratch. Hierarchies of hierarchies could even be constructed, thereby simplifying the layout process even more.

Another serious problem is the lack of an *undo* feature, which would let the user back out of the last modification made to either the circuit being edited or to an input waveform. This feature, common among many editing tools, would let the user recover from mistakes made while editing the circuit or waveform. For example, one common accident is to use the DELETE operation to remove a glue box instead of the UNGLUE operation. After accidentally deleting all the circuit elements inside the glue box, the user could simply click an undo button which would recover the circuit elements deleted during the last operation.

While the signal editor is convenient for a relatively small number of signals over a short duration, editing a greater number of signals having a longer duration can become cumbersome. For example, there is no way to easily insert or delete a signal value or a group of signal values. In the future, these deficiencies may be overcome by dumping the waveforms in tabular form to a text file which can then be edited by the user using a text editor. Once the user is finished editing the waveforms, the signal display would be updated with the contents of the modified text file.

Despite these restrictions, the GUI in its current manifestation provides a reliable, consistent and hopefully intuitive interface for digital circuit layout and simulation, especially in an academic setting. The current implementation provides a reasonable foundation upon which further enhancements can be based.

Chapter 3

GUI Implementation

This chapter provides insight into the internal workings of some of the important aspects of the GUI described in the previous chapter. In general, details regarding the code modularization, internal data representations and strategies adopted by the implementation will be described. In particular, information relating to the construction and manipulation of the components and netlists will be revealed. Several code fragments will be presented so as to illustrate various implementation techniques.

3.1 Overview of the Implementation

As mentioned in the previous chapter, the entire GUI was written using the scripting language Tcl/Tk. The high-level of abstraction offered by this language helped overcome many of the tedious, low-level implementation hurdles commonly encountered when developing a GUI. Also because there is no need to compile Tcl scripts, development is reduced to relatively simple iterations of edit $\Rightarrow$ execute cycles. Conventional compiled languages, such as C and C++, require much more time consuming edit $\Rightarrow$

compile ⇒ link ⇒ execute cycles.

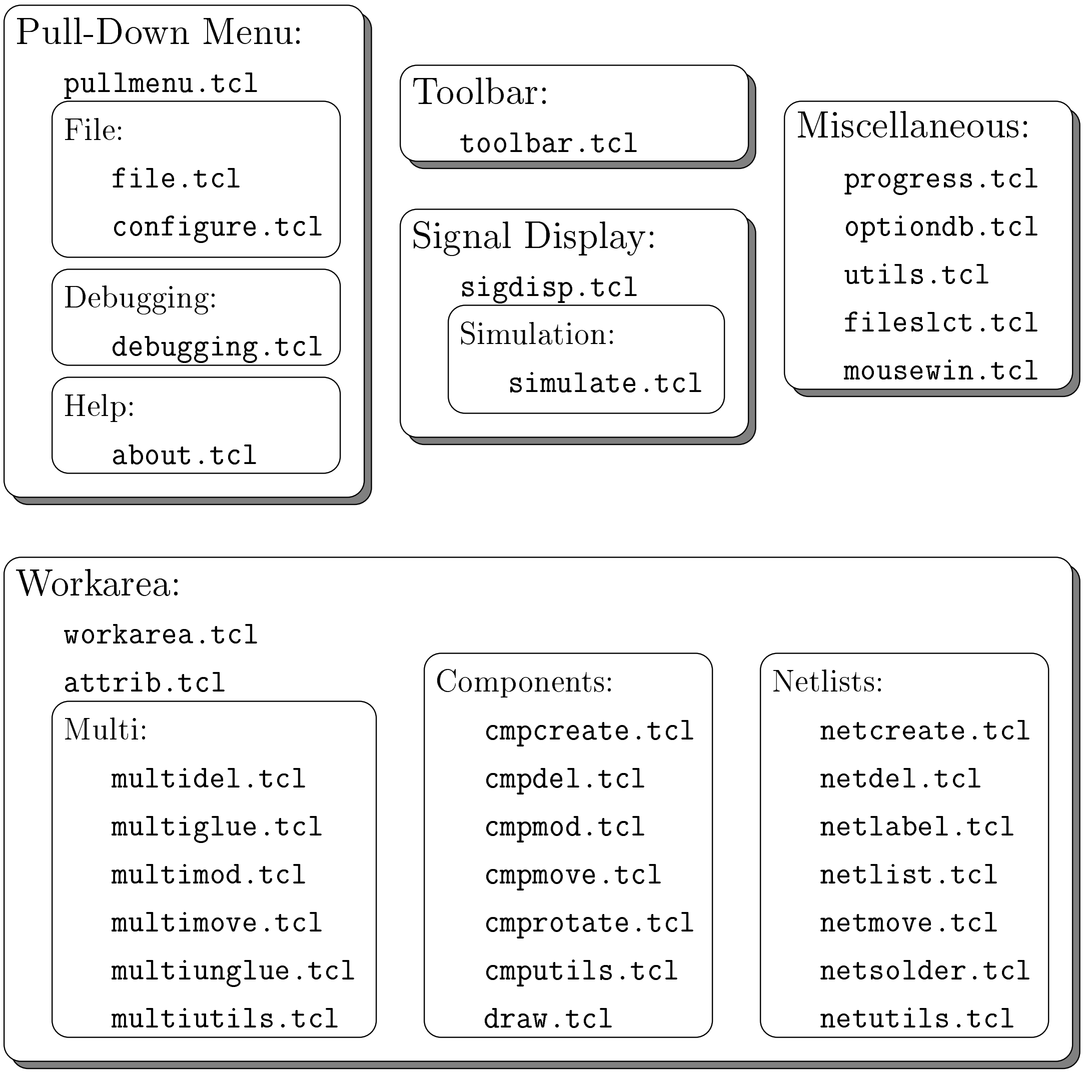
The GUI is comprised of approximately 7500 lines of Tcl/Tk code which includes about 2200 lines of internal documentation and 5300 lines of Tcl script. Using the "autoloading" feature of Tcl, the code was broken down into nearly three dozen sub-scripts or modules as shown in Figure 3.1. A main module named `tksim` is responsible for calling the appropriate procedures that create the circuit editor window and signal display window and basically serves to set the GUI in motion.

When a production release is made, all the scripts are combined into one large script file; all the comments are stripped out so as to reduce the overall size and increase the execution speed of the script.[1] In order to run the production release, several environment variables have to be set, which inform the GUI of the location of various bitmap files and the location of the simulator engine executable. More details regarding the installation of the production release is presented in Appendix A.

One of the main problems with Tcl is the lack of modular namespaces — the names of global variables or procedures in one module may conflict with the names of variables in other modules. To get around this problem, a naming scheme was adopted whereby the names of global variables and procedures were prefixed with unique strings which help classify the name and make the name unique. For example, variables and procedures which are directly related to components are prefixed with the string `comp_`; likewise, variables and procedures related to netlists are prefixed with `net_`. In order to encourage encapsulation, abbreviations of prefixes were used to indicated that a procedure or variable should only be used by other procedures

[1] Remember that Tcl is an interpreted language. Therefore any comments appearing inside loops, for example, have to be identified and ignored each time the loop is executed. By eliminating the comments, we eliminate this extra parsing.

Figure 3.1: Modularization of the GUI Source



inside the module in which it is defined. For example, the procedure `draw_rotate` in the `draw.tcl` module can be used by procedures in other modules, but the procedure `dr_rotate_item` should only be called within the `draw.tcl` module. This approach is not ideal since it is not enforced by the underlying language. However, it does help to overcome the lack of scope control in the global namespace.

Due to space constraints, it is not possible to describe in intimate detail every single procedure implemented by the GUI; therefore, only certain high-level aspects of the implementation will be elaborated upon in detail during this chapter. The code contains sufficient internal documentation, so the lack of external documentation in this report should not present too many problems to those wishing to explore and understand the internals of the package in greater detail.

3.2 Pull-Down Menu Modules

The modules responsible for implementing the pull-down menu and the options that appear in its submenus are presented in Table 3.1. The use of the `menubutton` and `menu` widgets facilitated the creation of the pull-down menu. The procedure dispatch mechanism is incorporated as a configuration option to entries added to the `menu` widget.

The pull-down menu user interface component is created by the `pullmenu_create` procedure of the `pullmenu.tcl` module. This procedure is generic enough to make the creation and subsequent modification of the menu bar and its corresponding pull-down menus relatively simple. An example of the invocation of this procedure is shown in Figure 3.2. This procedure accepts one list for each pull-down menu to be created. Each list is comprised of the name of the menu button, the location of the menu button along the menu bar (either `right` or `left`) and a list of menu entries which are to be displayed in the corresponding pull-down menu. Each of these entries are sublists which contain the necessary information to create the menu entry.

Note that the ampersand, `&`, may be used to indicate which character will be underlined in the text of the menu entry. This character becomes a *hot key* for

Table 3.1: Pull-down Menu Module Responsibilities

Module Name	Purpose
`about.tcl`	Display a top-level dialog box containing the name and version information of the application. Employed by the Help | About pull-down menu option.
`configure.tcl`	Displays a dialog box which lets the user configure various aspects of the GUI. Employed by the File | Configure... pull-down menu option.
`debugging.tcl`	Contains procedures used to help in the debugging and verification of various aspects of the GUI. These procedures are accessible via the Debugging submenu of the pull-down menu.
`file.tcl`	Contains procedures which implement several of the file related features of the File submenu of the pull-down menu.
`pullmenu.tcl`	Procedures responsible for the construction and manipulation of the toplevel pull-down menuing system.

that menu entry. Menu entries along the menu bar can be accessed by pressing **Alt** and the menu's corresponding hot key, thereby displaying the appropriate pull-down menu. An entry in a pull-down menu can be invoked by simply pressing the hot key corresponding to that entry. For example, the key sequence **Alt+F X** will terminate the application. The vertical bar, |, is used to separate submenu entries from their corresponding accelerator key sequence. This sequence enables the user to invoke a feature immediately, bypassing the pull-down menu system altogether. For example, **Ctrl+X** will also terminate the application, but with one less key stroke than using

Figure 3.2: Invoking the `pullmenu_create` Procedure

```
set menubar [pullmenu_create {
    {" &File " left
        {command "&New|Ctrl+N" file_new}
        {command "&Open...|Ctrl+O" file_open}
        {command "&Save...|Ctrl+S" file_save}
        {command "S&ave As...|Ctrl+A" file_save_as}
        {command "&Configure...|Ctrl+C" config_user_interface}
        {command "E&xit|Ctrl+X" file_exit}
    }
    ...
}]
```

the hot key sequence mentioned above.

Pull-down menu entries are not restricted to executing commands or procedures as implied by the figure. Entries may have radio- or check-buttons associated with them, thereby maintaining state information, or they may invoke other submenus (known as *cascades*), which offer more choices to the user when activated.

3.3 Toolbar Modules

All the procedures which construct and manipulate the toolbar are contained in one module, as shown in Table 3.2.

The toolbar, which is created by `toolbar_create`, is comprised of one large frame which encompasses all the buttons and labels contained within the toolbar. The buttons themselves are packed horizontally in smaller subframes. The organization of buttons on the toolbar is controlled indirectly by the associative array `tb_buttons` which determines how the buttons are grouped. Each element of the `tb_buttons` array contains a grouping of buttons which are represented by a list of sublists of

Table 3.2: Toolbar Module Responsibilities

Module Name	Purpose
`toolbar.tcl`	This module contains procedures for the creation and manipulation of the toolbar. Procedures to establish appropriate workarea canvas bindings upon button activation are also included in this module.

bitmaps which are displayed in the buttons. For example, the organization of the Components buttons in the toolbar presented in Figure 2.1 was achieved by the list grouping shown in Figure 3.3.

Figure 3.3: The `tb_buttons` Associative Array

```
set tb_buttons(Components)  {
    {cmpcreate_and      cmpcreate_nand}
    {cmpcreate_or       cmpcreate_nor}
    {cmpcreate_xor      cmpcreate_xnor}
    {cmpcreate_buffer   cmpcreate_not}
}
```

As with the pull-down menu, new buttons can be easily added to the toolbar by modifying the `tb_buttons` array; the bitmaps which appear on the buttons can be drawn using the X client *bitmap*. The location of the toolbar is controlled by the procedure `tb_move` which positions the toolbar either to the left or right of the workarea canvas.

In most cases, several procedures are invoked when a button is pressed. These procedures have two major responsibilities. First, they keep the button depressed until the user has either completed or aborted the chosen operation. Second, they establish appropriate canvas bindings on the workarea which carry out the operation selected from the toolbar when the leftmost mouse button is pressed on a circuit element which will be affected by the operation.

In accomplishing this second task, the procedure tb_set_bindings is called for each circuit element which is affected by the selected toolbar operation. An abbreviated version of that procedure is presented in Figure 3.4.

Figure 3.4: Establishing Canvas Bindings in tb_set_bindings

```
proc tb_set_bindings {item canvas status button} {
    if {$status == "on"} {
        $canvas bind $item <Enter> \
            "attrib_$item $canvas selected current"
        $canvas bind $item <Leave> \
            "attrib_$item $canvas normal current"
        $canvas bind $item <1> \
            "${item}_dispatch_${button} $canvas %x %y"
    } else {
        $canvas bind $item <Enter> {}
        $canvas bind $item <Leave> {}
        $canvas bind $item <1> {}
    }
}
```

The procedure will invoke the appropriate attrib_ procedure, described later, as the mouse pointer enters and leaves affected circuit elements. Pressing the leftmost mouse button, indicated by <1> in the code fragment, will dispatch the appropriate procedure to perform the operation on the circuit element currently under the mouse pointer.

The actual tb_set_bindings procedure contains some special cases which handle the rotation of components (which use both the leftmost and middle mouse buttons) and multiboxes.

3.4 Workarea Modules

The two modules related to the high-level operation of the workarea are presented in Table 3.3. A number of lower-level submodules related to components, netlists and multiboxes are also related to the workarea and are presented in subsequent sections.

Table 3.3: High-level Workarea Modules

Module Name	Purpose
`attrib.tcl`	Highlights and dehighlights workarea circuit elements (such as components and wires) as the mouse pointer enters and leaves them.
`workarea.tcl`	This module is responsible for the creation, configuration and manipulation of the workarea upon which the components and netlists are placed, modified and deleted by the user.

The workarea is created by the procedure `workarea_create` of the `workarea.tcl` module. Its basic duties are to create and position the workarea canvas and the horizontal and vertical scrollbars. It also dispatches another procedure to draw the grid lines on the canvas using the intensity specified in the user's configuration file. The `workarea.tcl` module also contains procedures which translate between screen and canvas coordinates and some data abstraction procedures which return the workarea widget and canvas widget for use by other modules.

The second module related to the workarea, `attrib.tcl`, changes the colour and width attributes of circuit elements as the mouse pointer enters and leaves them. This is done so that the user can easily determine which circuit element will be affected by the current operation. All the procedures contained in this module work basically the same way. The parameters to these procedures consist of the workarea canvas widget name, the highlighted status of the element (either `normal` or `selected`) and the tag or identifier of the circuit element to change. The procedures retrieve the desired colour and width attributes of the circuit element from the resource database and then use the `itemconfigure` canvas widget command to change the appearance of all the canvas primitives which comprise the circuit element. For some circuit elements, such as wires and points, this operation is simple, since they are comprised of only one canvas primitive. For more complicated elements such as gates, the operation must iterate over all the canvas primitives which comprise the element and change the width and colour attribute for each primitive.

A brief discussion of how structured graphics are created and manipulated using the Tk canvas is in order. Consider the short, but complete, code fragment presented in Figure 3.5. The first two lines simply create a canvas and place it on the display. Note that `-borderwidth` and `-highlightthickness` are called *widget options* and control the width of the canvas border and the thickness of the highlight focus ring surrounding the canvas respectively. In this particular case, they are both set to zero, meaning that the canvas will have no border and no highlight focus ring. The next two lines use the `create` canvas widget command to create two canvas items. These two commands create two short oblique line segments which intersect each other at their end points, forming a $\wedge$ in the upper left region of the canvas.

Figure 3.5: Creating Structured Graphics on a Canvas

```
canvas .c -borderwidth 0 -highlightthickness 0
pack .c
.c create line 10 0 0 20 -tags x
set id [.c create line 10 0 20 20 -tags x]
.c move x 40 40
.c itemconfigure x -fill blue -width 3
.c move $id -20 0
```

The `create` canvas widget command always returns a unique numeric identifier representing the canvas item just created, so that the canvas item may be referenced in the future. In our code example, the identifier of the first line is returned but discarded by our script. The identifier of the second line, however, is saved in the variable `id`, as it is used later. Also note that each of the line segments created are given the tag `x`, which may be used in the future to refer to both of these canvas items as a single entity.

The fifth line of the script moves all canvas items tagged with `x` 40 pixels down and 40 pixels to the right relative to their last position. Both lines are moved because they have both been tagged with the `x` tag when they were created. The sixth line then modifies the `-fill` and `-width` attributes of both lines to `blue` and `3` respectively using the `itemconfigure` widget command. This has the effect of colouring both line segments blue and increasing their width, making them easier to see on the canvas. Finally, the seventh line moves the canvas item identified by the contents of variable `id` (the second line segment created earlier) 20 pixels to the left. The end result of this script is that a $\vee$ will be displayed at an offset of 30 pixels horizontally and 40 pixels vertically from the upper left corner of the canvas widget.

Note that when this script is run, the user will see only the cumulative results of all the canvas operations; the creation of the $\wedge$ and its subsequent manipulation will not be seen. This is because Tk, for efficiency purposes, buffers all the screen updates until idle time is available to actually flush the updates to the display. To actually see the effect each line of the script has on the canvas, the code sequence "`update; after 1000`" can be inserted between each line. This causes Tk to flush the updates to the display immediately and to wait 1000 milliseconds (one second) so that the user may see the incremental effects of the script.

3.5 Component Modules

Components represent the elementary functional units of the circuit and are represented graphically on the workarea canvas by logic gates. The implementation modules related to the construction and subsequent manipulation of components are presented in Table 3.4.

3.5.1 Component Creation

When the user clicks on a component button in the toolbar, a workarea canvas binding is established which will result in the creation of the selected component when the mouse pointer enters the canvas. The components themselves are composed of several canvas primitives, such as lines, arcs, rectangles and ovals all of which are tagged with an identical tag. This enables all the primitives to be treated as a single entity by

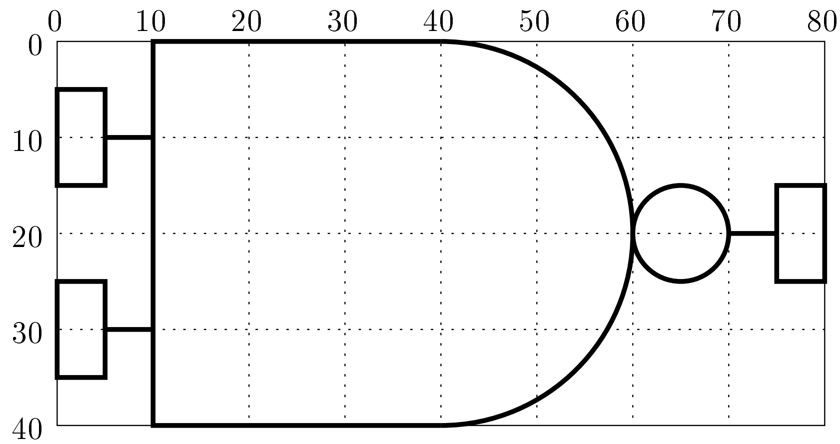
Table 3.4: Component Creation and Manipulation Modules

Module Name	Purpose
`cmpcreate.tcl`	This module is responsible for drawing and placing the components on the workarea canvas. Uses the `draw_component` procedure in the `draw.tcl` module.
`cmpdel.tcl`	Module to delete components from the canvas. Also updates the netlist arrays by removing deleted port assignments.
`cmpmod.tcl`	Modify the attributes associated with a component. Currently, no attributes are supported; but in the future, component transport delay will be provided.
`cmpmove.tcl`	Move components on the canvas. Procedures must also move any wires and points attached to the ports of the component.
`cmprotate.tcl`	Module to rotate components on the canvas. Employs the `draw_rotate` procedure in the `draw.tcl` module.
`cmputils.tcl`	Miscellaneous low-level helper procedures related to components, such as overlap detection and identification.
`draw.tcl`	Contains the low-level code and helper procedures required to draw and rotate components on the canvas. Also contains code to determine the bounding box of a canvas item.

the implementation.

For example, consider the construction of the *NAND* gate presented in Figure 3.6. The two input ports and one output port are rectangles; the three wires leading from the ports to the component body are single line segments; the component body adjacent to the two input ports is comprised of three line segments; the component body adjacent to the output port is an arc and the *NOT* circle is an oval primitive. Note that it is important for the midpoint of the outside edge of a port rectangle to coincide with the intersection of two grid lines. This is so wire points, which are also created at intersecting grid lines, may be soldered to the port.

Figure 3.6: Primitive Composition of a *NAND* Gate



These nine primitives are all tagged with a unique alphanumeric name generated by the procedure `dr_generate_tag`. The tag consists of the string `comp`, followed by the type of the gate which, in turn, is followed by the serial number for that gate. All these elements of the tag are separated by underscores. Therefore all the primitives comprising the eighth *NAND* gate will be tagged with the string `comp_nand_8`.

The Tcl procedures which build all the gates are contained in the `draw.tcl` module. As an example of such a procedure, consider the code fragment in Figure 3.7. The `colour` and `width` variables are all determined from the resource database. The `dr_wires_not` procedure draws and tags all the wires connecting the ports to the component body and the *NOT* circle at the end of the gate. The `dr_ports` procedure draws the ports of the component; input ports are represented by green rectangles and output ports by red rectangles. The ports of a component are given additional tags to identify them as special parts of a component which can be connected to netlists. The parameters of both of these procedures consist of the component tag and the coordinates of the primitives they are to construct.

Note that all the canvas primitives which make up a component are given the tag `component`. This tag permits the implementation to quickly distinguish a component primitive from a netlist primitive on the canvas and lets the implementation identify components during operations which have components as their targets.

Figure 3.7: Tcl Code to Build a *NAND* Gate

```
.c create line 40 0 10 0 10 40 40 40 \
    -fill $colour -tags "component $tagname" -width $width
.c create arc 20 0 60 40 -start 270 -extent 180 -style arc \
    -fill $colour -tags "component $tagname" -width $width
dr_wires_not $tagname \
    {{10 10 5 10} {10 30 5 30} {70 20 75 20}} {60 15 70 25}
dr_ports $tagname \
    {input {0 5 5 15} {0 25 5 35}} {output {75 15 80 25}}
```

In order to reduce code duplication, one procedure is responsible for producing both *NAND* and *AND* gates, depending upon its parameters. Therefore, the actual source code for constructing gates is a little more complicated than that presented in Figure 3.7. Currently, the interface is limited to creating the eight gates mentioned in Chapter 2. In order to extend the GUI to support other gates or higher level devices such as decoders and multiplexers, the `draw.tcl` module will have to be modified accordingly to support the construction of the new devices using the canvas primitives.

Upon construction of the gate, it is immediately moved from its origin in the upper left of the canvas to the current location of the mouse pointer on the canvas. Because canvas updates are buffered, the user will not notice the gate being drawn initially in the upper left corner of the canvas before it is moved to the current mouse location. A binding is established so that the motion of the mouse will cause the component to move across the canvas. The GUI employs gridded placement so that the component will move in discrete steps across the canvas so as to preserve their alignment with the canvas grid lines. This makes it easier to place and align the components. The synchronous motion of the component with the mouse is broken when the user presses the leftmost mouse button which affixes the component to its current position on the canvas.

3.5.2 Component Representation Arrays

The associative array `draw_params` is used to store various attributes related to a component. It is indexed by a component tag and a component attribute. The attributes currently supported are `canvas`, `colour`, `width` and `orient`. For example, if the orientation of the component with tag `comp_nand_8` is currently 180 degrees, then the value of the array element `draw_params(comp_nand_8,orient)` will be `180`. The `canvas` attribute indicates in which canvas the component resides. This attribute exists for future extensibility of the GUI, in which multiple views of a circuit over several different canvases may be possible. The `colour` attribute stores the colour to be used when drawing the primitives; by default, they are black. The `width` attribute indicates the outline thickness to be used when drawing the primitives which make up the component; by default the widths are set to one pixel. The default values for the initial colour and width of a component can be changed using the option database. The `orient` attribute, as alluded to above, stores the current rotation of the component. Currently, the orientation of a component must be one of `0`, `90`, `180` or `270`. The purpose of the `draw_params` associative array is to cut down on the number of global variables required by the `draw.tcl` module and to reduce the number of parameters which must be passed among procedures.

Another array, `cmp_coord` is used to store the location of the component on the canvas. It too, is indexed by the unique tag name of the component and contains the canvas coordinates of the center of the component. In retrospect, the `cmp_coord` array could have been merged with the `draw_params` array, so as to limit pollution of the global namespace.

3.5.3 Component Manipulation

Components are manipulated by selecting an operation from the toolbar and then clicking the leftmost mouse button on the target component. In order to make the selection of the component easier for the user, transparent bounding boxes are placed on top of components when they are created. These bounding boxes make components easier to grab by providing a greater area on the canvas which can be clicked on by the user. Without these bounding boxes, the user would have to click on the thin outline of the primitives of the component in order to perform the desired operation. Each component has two types of bounding boxes, each of which are tagged with the unique tag name of the component as well as special tags identifying them as bounding boxes. One bounding box covers the entire component body and its ports, whereas the second bounding box only encompasses the component body. Together, these two bounding boxes prevent components from overlapping with one another and ensure that if a wire overlaps a component, it does so only at the ports of the component.

Moving and Deleting Components

Deleting and moving components is relatively simple using Tcl/Tk. The only difficult aspect is updating any wires which are connected to the component. When the user selects a component to move or delete, all the wire points which are attached to the ports of the components are determined using the component utility procedure `cmp_get_point_ids`. If the user is deleting the component, the internal netlist representation is updated using the netlist utility procedure `net_list_remove_port` which, in effect, disconnects the ports of the component from their respective netlists. The canvas `delete` command is then used to remove the component from the workarea. If the user is moving a component, then a binding is established which calls the canvas `move` command to adjust the location of the component as the user moves the mouse. The end points of any wires which are connected to the ports of the component are moved in harmony with the component itself using the netlist utility procedure `net_do_move_point`.

Rotating Components

Rotation of components presented a challenge since Tcl/Tk does not directly support the rotation of canvas primitives. Therefore, each primitive which comprises a component had to be rotated individually using rotation matrices represented mathematically by the equations shown in Figure 3.8. The `xp` and `yp` variables represent the coordinates of the primitives as determined by the canvas `coords` command while `xc` and `yc` represent the center of rotation, which is simply the center of the component as determined by the `cmp_coord` array.

Figure 3.8: Rotation Equations for Canvas Primitives

```
set dr_roteq(x,0)   { expr $xp }
set dr_roteq(y,0)   { expr $yp }
set dr_roteq(x,90)  { expr $xc + $yc - $yp }
set dr_roteq(y,90)  { expr $xp + $yc - $xc }
set dr_roteq(x,180) { expr 2 * $xc - $xp }
set dr_roteq(y,180) { expr 2 * $yc - $yp }
set dr_roteq(x,270) { expr $xc - $yc + $yp }
set dr_roteq(y,270) { expr $yc + $xc - $xp }
```

The procedure `draw_rotate` is the main procedure responsible for rotating a component. This procedure simply iterates over each of the primitive canvas items of a component and invokes the procedure `dr_rotate_item` which makes use of the transformation equations to rotate each of the primitives. The arc primitive present in the *AND* and *NAND* gates had to be treated as a special case by this procedure. As with deleting and moving a component, the wires connected to a component have to be adjusted so as to stay connected to the ports of the component.

3.6 Netlist Modules

In the context of this application, a netlist is defined as a collection of wires and points which are logically equivalent during the course of a simulation. The modules which create and maintain the netlists in the workarea canvas are presented in Table 3.5.

Table 3.5: Netlist Creation and Manipulation Modules

Module Name	Purpose
`netcreate.tcl`	Contains all the procedures necessary for the creation of wires and wire points on the workarea. Supports rubber banding as well as gridding of wires and wire points.
`netdel.tcl`	Procedures for deleting wire points and wires.
`netlabel.tcl`	Procedures responsible for displaying a dialog box for the purposes of letting the user label a point on a netlist.
`netlist.tcl`	This module contains several miscellaneous procedures which manipulate netlists as a whole. Operations such as identifying, splitting and uniting netlists are contained in this module.
`netmove.tcl`	Procedures for moving wires and wire points.
`netsolder.tcl`	Contains all the procedures necessary for soldering netlists together. Essentially, these procedures connect overlapping ports, points and wires together.
`netutils.tcl`	Several miscellaneous procedures which create and manipulate individual wires and wire points.

3.6.1 Netlist Creation

After selecting the netlist CREATE button from the toolbar, a procedure named `net_set_point` is invoked to create a wire point each time the user presses the leftmost mouse button on the workarea canvas. Upon creation of the wire point, represented on the workarea by a polygon primitive in the shape of a small diamond, a wire segment is drawn from the wire point to the current position of the mouse pointer. A binding is created which causes the end point of the newly created wire to follow the motion of the mouse, effectively "rubber banding" the wire. Wire points are snapped to the nearest intersecting grid lines on the workarea so as to make it easier to align circuit elements with one another.

Before actually laying down a wire point, `net_set_point` first ensures that the point is not being placed inside the body of a component. Then, if the user interface has been configured to perform automatic soldering, the procedure will also attempt to solder the newly created point to any ports, wires or other wire points that overlap with its position. The `net_set_point` procedure maintains two lists containing the canvas identifiers of the wires and wire points created during the current netlist creation operation. These lists are consulted to delete the last wire and wire point from the workarea canvas should the user press the **BackSpace** key. Likewise, these lists are also used to delete all the wires and wire points created should the user decide to abort the create operation by pressing the **Esc** key or rightmost mouse button.

Netlist elements consist of wire segments and wire points and each are tagged with the names `wire` and `point`, respectively. Both netlist elements also receive the tag `netlist` which serves to distinguish them from components. In addition, all netlist elements which comprise a single netlist are tagged with a unique name which serves to identify the netlist as a single entity. The format of this netlist tag name consists of the string `netlist_` followed by the sequence number of the new netlist. This is similar to the tagging mechanism used for components. For example, all the wires and wire points of the fourth netlist created by the user would be tagged with the string `netlist_4`. Netlist tag names are generated by the procedure `net_list_generate_id`.

Dynamic Netlist Identification

By tagging each element of a netlist with a unique name, netlist extraction becomes much easier and faster. This feature, known as *dynamic netlist identification*, involves the tagging and retagging of netlist elements as they are created and modified by the user so as to reflect the new netlist connectivity. As the user creates a new netlist, the points and wires comprising the netlist are tagged with the unique netlist name. If the user solders two netlists together or splits two netlists apart, then the netlist tags are updated accordingly. In effect, netlists are "extracted" while they are being created and modified, thus eliminating the need for expensive netlist traversals each time the user wishes to simulate the circuit. Further details regarding the soldering and splitting of netlists and the subsequent retagging are presented in a later subsection.

In addition to permitting easier netlist extraction, dynamic netlist identification also serves to help enforce many of the restrictions with respect to netlist layout. Because such restrictions can be identified quickly, the GUI can prohibit potential layout errors interactively. For example, using dynamic netlist identification, the GUI can determine in constant time whether or not the user is attempting to introduce a cycle in a netlist.

Using dynamic netlist identification, it is very simple to implement a feature which will let the user see all the wires and points which comprise a single netlist. Users can highlight an entire netlist by using the NETLIST toolbar button. This procedure creates a binding which is triggered whenever the user moves the mouse pointer inside a netlist element, as shown previously in Figure 3.4. The implementation extracts the unique netlist tag name from the netlist element and changes the colour attribute of all the items on the canvas having that tag name. The netlist is dehighlighted when the mouse pointer leaves the netlist element.

3.6.2 Netlist Representation Arrays

In order to internally represent the netlists and their corresponding component connectivity, the Tcl/Tk script employs several global associative arrays. The array names, their indices and their contents are shown in Table 3.6. Notice that all associative arrays which are indexed by point identifiers are prefixed with `pnt`, whereas associative arrays indexed by netlist tag names are prefixed with `net`.

As an example of how netlists are represented internally, consider the incomplete circuit displayed in Figure 3.9. Remember that all newly created canvas items are assigned unique numeric identifiers so that they may be referenced later on. However, for clarity, capital letters will be used to represent point identifiers, integers will represent the identifiers of wire segments and lowercase letters will represent port identifiers (that is, the identifiers of port rectangles). A shaded port indicates that the port has been connected to the overlapping point. The sans serif font represents netlist labels placed on the workarea by the user via the Netlist Label dialog box; the

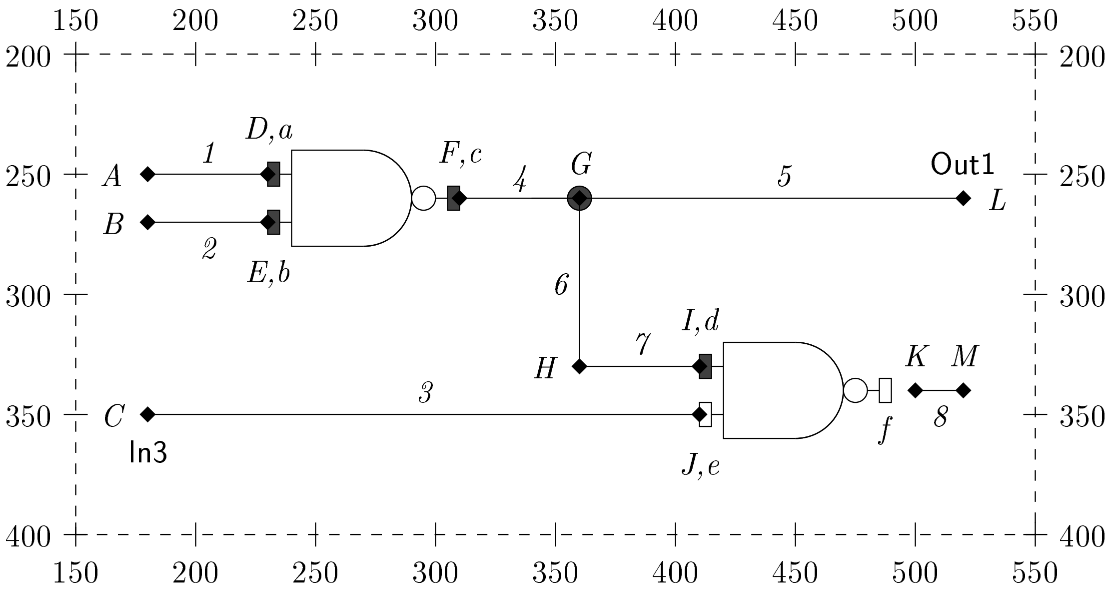
Table 3.6: Netlist Representation Arrays

Array Name	Array Index	Array Contents
pnt_coord	point identifier	(x, y) coordinates of the point.
pnt_wires	point identifier	List of wire identifiers adjacent to the point.
pnt_port	point identifier	Canvas identifier of the port connected to the point (if applicable).
pnt_label	point identifier	Canvas identifier of the textual label for the point (if applicable).
net_ports	netlist tag name	List of ports connected to a netlist. Each element of the list is a pair consisting of a port type and the point identifier connected to the port.
net_name	netlist tag name	The textual name of the netlist as assigned by the user via the Netlist Label dialog box.

numeric identifier assigned to this text label by Tk will be represented by the notation id(label).

The tags for each of the points and wires for this circuit are shown in Tables 3.7 and 3.8 respectively. Note that all points have been given the `point` tag and all wires have been given the `wire` tag. Also note that in addition to the `netlist` tag, all wires and points have been given a special netlist identifier tag name which serves to indicate to which netlist the wire or point belongs.

Figure 3.9: Example of a Circuit Layout



Point Arrays

The four point arrays representing the circuit are shown in Table 3.9. The purpose of the `pnt_coords` array is fairly straightforward; it allows the implementation to quickly access the coordinates of the center of the diamond-shaped polygon used to represent wire points on the canvas. By storing these coordinates, the implementation relieves itself from having to recalculate the center of the point based upon the canvas coordinates of the polygon. It also makes the implementation more resilient to change should a different shape be used to represent the wire points.

The `pnt_wires` array contains the identifiers of all the wires adjacent to a given wire point identifier in the netlist. Consequently, the degree of a point can be determined by simply finding the number of elements in the `pnt_wires` list that corresponds to the point. In Tcl, this can be accomplished with a call to the `llength` list procedure. For example, `$pnt_wires(`G`)` is {`4 5 6`}; hence `llength $pnt_wires(`G`)` is

Table 3.7: Point Tags for the Circuit in Figure 3.9

Point Identifier	List of Tags
A	{ `point netlist netlist_0` }
B	{ `point netlist netlist_1` }
C	{ `point netlist netlist_2` }
D	{ `point netlist netlist_0` }
E	{ `point netlist netlist_1` }
F	{ `point netlist netlist_3` }
G	{ `point netlist netlist_3` }
H	{ `point netlist netlist_3` }
I	{ `point netlist netlist_3` }
J	{ `point netlist netlist_2` }
K	{ `point netlist netlist_4` }
L	{ `point netlist netlist_3` }
M	{ `point netlist netlist_4` }

three, so point G has degree three.

The `pnt_port` array is used to associate a point with a component port when the two have been connected by the user. This array is consulted when the user moves a point which is connected to a component port. In this situation, the array maps the point identifier to the corresponding port identifier to which it is connected. From this, the implementation can determine the unique tag name of the component which is to be moved in unison with the point. The array is also used to map identifiers in the reverse direction when the user moves a component attached to one or more

Table 3.8: Wire Tags for the Circuit in Figure 3.9

Wire Identifier	List of Tags
1	{ `wire netlist netlist_0` }
2	{ `wire netlist netlist_1` }
3	{ `wire netlist netlist_2` }
4	{ `wire netlist netlist_3` }
5	{ `wire netlist netlist_3` }
6	{ `wire netlist netlist_3` }
7	{ `wire netlist netlist_3` }
8	{ `wire netlist netlist_4` }

wire points. The procedure `net_port_point` does this mapping by linearly scanning the elements in the `pnt_port` array. Notice that if a point is not connected to a component port then the `pnt_port` element corresponding to the point is undefined. Also, if a component is deleted, then the appropriate elements of the `pnt_port` array must be updated to "disconnect" the points from the ports of the deleted component.

The `pnt_label` array provides an easy way for the implementation to quickly access the identifier of the textual label canvas item which names a point and its corresponding netlist. Given the identifier of the point, the text label canvas item associated with that point can be obtained, thereby allowing the label to be subsequently modified or deleted by the implementation. Note that because each netlist can only have one name, the number of elements set in the `pnt_label` array cannot exceed the number of netlists in the circuit. With respect to the sample circuit of Figure 3.9, there are five netlists, therefore, the `pnt_label` array can have no more

Table 3.9: Point Array Values for the Circuit in Figure 3.9

Point Identifiers	`pnt_coords`	`pnt_wires`	`pnt_port`	`pnt_label`
A	180 250	*1*	–	–
B	180 270	*2*	–	–
C	180 350	*3*	–	id(In3)
D	230 250	*1*	a	–
E	230 270	*2*	b	–
F	310 260	*4*	c	–
G	360 260	*4 5 6*	–	–
H	360 330	*6 7*	–	–
I	410 330	*7*	d	–
J	410 350	*3*	–	–
K	500 340	*8*	–	–
L	520 260	*5*	–	id(Out1)
M	520 340	*8*	–	–

than five elements set. Also, if one point in a netlist has been given a label, then no other point in that netlist may be labelled. For example, because wire point L already has a label (`Out1`), the points F, G, H and I; which belong to the same netlist as L, cannot be given labels. Consequently, the elements in the `pnt_label` array corresponding to these points are not set. If none of the points in a netlist have their corresponding `pnt_label` set, then the netlist is unnamed. This is the case with the netlist determined by the points A and D.

There are times when the implementation needs to know the identifiers of the two points at the end of a given wire segment. During early development, this was achieved using an array which was indexed by a wire identifier and stored the two point identifiers associated with the wire. However, this became too clumsy to maintain because this array would have to be kept consistent with the contents of the `pnt_wires` array. Therefore a simple auxiliary procedure named `net_wire_points` was written which accepts a wire identifier and scans the `pnt_wires` array to determine the identifiers of the two end points of the wire. Although this method is slower, the code became much more maintainable. A similar associative array, which was indexed by a port identifier and returned the identifier of the wire point connected to it (if the port was connected), was replaced with the procedure `net_port_point`, as described above.

Netlist Arrays

The netlists arrays representing the circuit are shown in Table 3.10. This table also includes a column showing the points associated with each of the netlists. Notice, however, that these points are not stored in any array by the implementation. Instead, as described earlier, all the elements comprising a netlist (wire segments and wire points) are each tagged with a unique tag name as they are created and manipulated.

The `net_ports` array, which maintains a list of port/point pairs for each netlist entry, is used primarily to determine the netlist type. Using the netlist type, the GUI can prevent the user from making mistakes during the design of the circuit. For example, netlist types are used to prevent the user from soldering two output netlists together and from directly modifying the signal values of an output netlist on the signal display. The type of a netlist is determined by the following rules:

Table 3.10: Net Array Values for the Circuit in Figure 3.9

Netlist Tag Name	Points in Netlist	`net_ports`	`net_name`
`netlist_0`	A D	{`input` D}	–
`netlist_1`	B E	{`input` E}	–
`netlist_2`	C J	–	In3
`netlist_3`	F G H I L	{`output` F} {`input` I}	Out1
`netlist_4`	K M	–	–

1. If the netlist is isolated (that is, none of its points are connected to ports) then the netlist type is *unknown*. For example, `netlist_2` and `netlist_4` from the previous example are of *unknown* type.

2. If the netlist is connected to one or more input ports and is not connected to any output ports, the netlist type is *input*. For example, `netlist_0` and `netlist_1` are input netlists.

3. If the netlist is connected to an output port then the netlist type is *output*. Note that in this case, it is irrelevant how many connections, if any, the netlist has to input ports. In the example, `netlist_3` is the only output netlist.

In order to implement the above rules, the `net_ports` array maps a given netlist tag name to a list of port/point pairs which represent the port types to which the netlist is connected and their corresponding point identifiers. Whenever this list is modified in any way, it is re-sorted so that an output pair, if one exists in the list, is always placed at the beginning. This way, the type of the netlist is always kept at the front of the list and can therefore be extracted in constant time. If the netlist has no corresponding `net_ports` array element, then the type will be *unknown*.

The `net_name` array, which contains the textual name of the netlist as seen by the user, serves as a liaison between the circuit editor and the signal display window. This name serves as a key which is used to identify signals waveforms in the signal display window. Like the `net_ports` array, the `net_name` array is also used to prevent the user from making mistakes during circuit layout. For instance, this array is consulted in order to prevent the user from trying to give a single netlist two names and from soldering together two netlists which have both been previously labelled. The `net_name` array is also used to prohibit the user from giving two separate netlists the same name.

Note that it is acceptable for a netlist to have no corresponding entry in either the `net_ports` or the `net_name` array. In this case, the netlist is simply an isolated netlist which has not been labelled by the user.

3.6.3 Netlist Manipulation

Like components, the wires and points of netlists can be manipulated by selecting the desired operation from the toolbar and then clicking the leftmost mouse button on the target wire or point. This section will describe how the GUI implements moving, deleting and soldering of netlist elements.

Moving Netlist Elements

The user may move wires or wire points by activating the MOVE button on the toolbar and clicking on the desired item and dragging it across the canvas. If the user clicked on a point, then the procedure `net_prepare_point_move` is invoked which determines if the point is connected to a component port. If it is, then the procedure prepares the component attached to the point for simultaneous movement; otherwise, only the point itself will be moved. If the user clicked on a wire, then the procedure `net_prepare_wire_move` is invoked. This procedure determines the identifiers of the two end points of the wire and ensures that the end points of the wire are not attached to any ports. If the wire is attached to a component, then the GUI will not permit the user to move the wire.[2]

The motion of the mouse is then bound to the procedure `net_do_move` which moves the selected wire or point by calling the utility procedure `net_do_move_point`. This procedure moves a point, and any labels attached to it, while simultaneously stretching or shrinking all wires to which the point is connected. When moving a wire, `net_do_move` calls `net_do_move_point` twice every time the mouse is moved — once for each of the two end points of the wire.

Deleting Netlist Elements

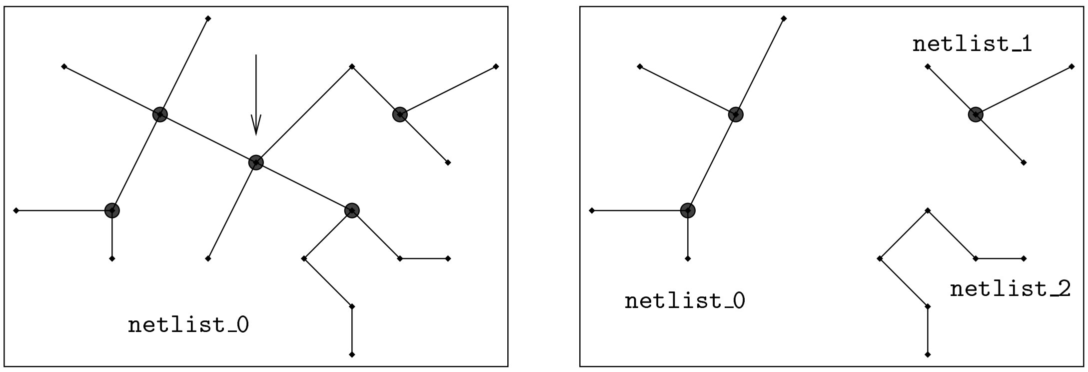
Deletion of netlist elements is more complicated due to the variety of potential circumstances which may arise. The procedure `net_delete_point` takes care of point deletion by first determining if the point to be deleted has degree two. If so, then the point and its adjacent wires will be deleted and the two points adjacent to the deleted point will be joined together. Note that deleting a point of degree two does not change the netlist connectivity; therefore, the netlist tags remain unchanged. If the user is deleting a point with degree one or degree greater than two, then

[2]Moving a wire connected to a component is problematic since the implementation would have to consider the possibility of moving *two* separate components if the wire is connected to both components. This situation has yet to be implemented by the GUI.

the `net_delete_point` procedure will cycle through all the wires connected to the point and delete them. In addition, the identifiers of the deleted wires are removed from the entries of the `pnt_wires` array corresponding to the points adjacent to the deleted point. If an adjacent point had degree one before the deletion, then that point will be removed. The deletion of a wire is handled similarly by the procedure `net_delete_wire`.

The deletion of a wire point or a wire which connects two or more netlists results in netlist splitting. Consequently, retagging of the netlist elements must occur so as to implement the dynamic netlist identification strategy. In order to retag the split netlists, a simple traversal algorithm was written which is called for each new netlist created as a result of the split. For example, in Figure 3.10, consider the deletion of the point indicated by the arrow in the netlist on the left. After the point is deleted, the netlists shown on the right are produced, each of which have had their respective elements uniquely retagged. Note that the deletion of the point caused the deletion of an adjacent point which had a degree of one.

Figure 3.10: Splitting a Netlist



Before Point Deletion After Point Deletion

The retagging of the netlist elements in each of the resulting netlists is performed by the procedure `net_list_traverse` as shown in Figure 3.11. The arguments to this procedure include the identifier of the point at which the traversal is to start (`pid`) and the old and new netlist tag name (`nid1` and `nid2`, respectively). The procedure performs a depth-first traversal starting at the supplied point and retags all netlist elements that had the tag `nid1` with the new tag `nid2`.

Figure 3.11: Traversing and Retagging a Netlist

```
proc net_list_traverse {canvas pid nid1 nid2 done} {
    upvar $done visited
    global pnt_wires

    if {$pid == ""} {
        return
    }
    set visited($pid) 1
    $canvas dtag $pid $nid1
    $canvas addtag $nid2 withtag $pid

    # Adjust 'pnt_port', 'pnt_label', 'net_name'
    # and 'net_ports' arrays.  (Not shown here.)

    foreach wid $pnt_wires($pid) {
        $canvas dtag $wid $nid1
        $canvas addtag $nid2 withtag $wid
        set adjpnt [net_get_adjacent_point $pid $wid]
        if {! [info exists visited($adjpnt)]} {
            net_list_traverse $canvas $adjpnt \
                     $nid1 $nid2 visited
        }
    }
}
```

Note that traversing a netlist after it has been split is somewhat expensive, especially considering the interpretive nature of Tcl. However, it was discovered during development that occasionally traversing and retagging a subset of the netlists was preferable over continually extracting *all* the netlists each time the user wanted to highlight netlists or simulate the circuit. Such extractions almost inevitably lead to a traversal of all the netlists.

Soldering Netlist Elements

The soldering operation results in an action opposite to that performed by the deletion operation — instead of splitting netlists, soldering unites two or more netlists. Uniting two netlists is much simpler to implement than splitting two netlists. To unite two netlists, one of the netlists is retagged with the tag name of the other. For example, to unite netlists `net1` and `net2`, we simply assign the tag `net2` to all netlist elements which have the `net1` tag and then remove the `net1` tag. This can be performed by a two line procedure as shown in Figure 3.12.

Figure 3.12: Source Code Uniting Two Netlists

```
proc net_list_unite {canvas nid1 nid2} {
    $canvas addtag $nid2 withtag $nid1
    $canvas dtag $nid1

    # Adjust 'net_name' and 'net_ports' arrays
    # as appropriate. (Not shown here.)
}
```

The main procedure responsible for soldering netlists is `net_start_solder`. In order to solder netlists, the procedure identifies all the points, wires and ports which overlap with the location where the soldering iron was activated on the workarea canvas. This list of items is sorted with points first, ports second and wires last.

Then, a loop is executed in which an attempt is made to solder the first item in the list with each of the successive list items via calls to procedures of the form `net_connect_*_to_*` where `*` is one of `point`, `port` or `wire`, depending upon the types of the two items being soldered. A point identifier representing the junction point for the soldering is returned by the `net_start_solder` procedure.

As mentioned earlier, the implementation prohibits the user from performing solderings that will create cycles or other inconsistencies in the netlists. Most of these consistency checks can be done quickly because of the dynamic netlist identification feature. Furthermore, as a visual aid to the user, the implementation fills in the ports of components when they are soldered to a point. Inputs ports are filled in a green colour and output ports are filled red. In addition, if the resultant soldering point has a degree greater than two, the implementation creates a soldering dot (●) at the junction point to help express the connectivity. This dot will be deleted if the degree of the point falls below three during subsequent manipulation of the netlist.

3.7 Multibox Modules

Multiboxes, which are synonymous with the *glue* boxes mentioned in Chapter 2, let the user group together several circuit elements and manipulate them as a single entity. The modules which implement multiboxes are shown in Table 3.11.

Multiboxes are displayed as rectangle outlines on the workarea canvas; they are created with the GLUE button on the toolbar. After the user presses this button, a binding is created which calls the procedure `multi_create_new` when the user clicks the leftmost mouse button on the canvas. This procedure creates an anchor point on

Table 3.11: Multibox Creation and Manipulation Modules

Module Name	Purpose
`multidel.tcl`	Module to delete a multibox and all the circuit elements contained within.
`multiglue.tcl`	Create the multibox on the screen by letting the user stroke out a bounding box around all the circuit elements to be contained within the multibox.
`multimod.tcl`	Module to modify the size of a multibox on the canvas.
`multimove.tcl`	Moves all the circuit elements contained within a multibox.
`multiunglue.tcl`	Removes the multibox from the canvas. This module does not delete the items within the multibox.
`multiutils.tcl`	Miscellaneous utility procedures for the manipulation of multiboxes.

the workarea canvas for the multibox. Another binding is then established between the motion of the mouse and the procedure `multi_do_set` which extends the multibox from the anchor point to the current position of the mouse. When the user releases the mouse button, the creation of the multibox is completed by invoking the procedure `multi_stop` which deletes the multibox if it had no area.

Resizing the rectangle of an existing multibox is performed by the MODIFY toolbar button. The implementation will identify mouse clicks on either the edges or corners of an existing multibox. If the user clicked on an edge, then the other three edges will act as anchors and the selected edge may be dragged horizontally or vertically. If the user clicked on a corner, then the opposite corner will act as an anchor and the selected corner may be moved diagonally. During modification of multiboxes, the same two procedures described above are reused so as to reduce code duplication.

When the user selects either the MOVE, DELETE or UNGLUE button from the toolbar, the implementation must consolidate all circuit elements enclosed within bounding boxes on the workarea so that each of the multiboxes and their contents may be treated as a single entity. The GUI accomplishes this by creating filled transparent rectangles on top of each of the bounding outline rectangles currently on the workarea canvas. As a result, whenever the mouse pointer moves over a multibox during one of these three operations, the entire multibox outline will be highlighted instead of the individual circuit elements contained within. Hence, the circuit elements contained inside the multibox cannot be directly accessed by the user.

Moving all the circuit elements in a multibox is achieved by tagging all enclosed components and multiboxes with the name `multi_tag` and by adding all enclosed wire points to a list of points. A binding is then established which binds the motion of the mouse with the movement of all these tagged circuit elements and the wire points in the point list. By moving the points individually, the wires corresponding to these points will also be moved. If a wire straddles a multibox, then one end point will move, whereas the other will remain stationary, thereby creating a rubber banding effect. During the implementation of multibox movement, several issues had to be resolved regarding circuit elements which do not lie entirely inside the selected multibox. For example, if a point attached to a component port lies inside the multibox, but the component itself straddles the outline of the multibox, then the point will not be moved by the GUI.

Deletion of a multibox and its contents is handled by iterating over each of the circuit elements contained inside the selected multibox and then invoking an appropriate delete procedure to remove the element from the display. The unglue feature is trivially implemented by deleting the selected multibox rectangle and its corresponding transparent rectangle from the canvas, leaving the enclosed circuit elements unaltered.

3.8 Signal Display Modules

The signal display modules are responsible for creating and manipulating the waveforms which are presented in the signal display window. Each labelled netlist of the circuit is represented by exactly one waveform in this window. The modules responsible for its implementation are outlined in Table 3.12.

Table 3.12: Signal Display Modules

Module Name	Purpose
`sigdisp.tcl`	Contains procedures to create and manipulate the signal display window. This module also contains several utility procedures related to the signal display window which can be invoked by other modules.
`simulate.tcl`	This module is responsible for transmitting the signal inputs and circuit description to the simulator engine for simulation. The resultant output signals are then read and displayed in the signal display window.

The Tk canvas widget was used to implement the signal labels, signal times and signal waveforms subwindows of the signal display window. By using canvases, the implementation was able to take advantage of the flexibility and accuracy of the placement of the contents of these windows, thereby achieving ideal alignment and synchronization. The implementation of the `sigdisp.tcl` module is too lengthy to describe in detail in this chapter; therefore, for brevity, only the procedural interface of this module will be discussed.

The `sig_draw_signal` procedure is responsible for displaying a signal and takes, as parameters, the signal name and a list containing time/value pairs; the latter parameter is optional. If the procedure is invoked with only the signal name then the duration of the signal and its values will be set according to pre-defined defaults, as established in the resource option database. If the calling procedure supplies a list of time/value pairs, then a waveform that corresponds with the elements of the list will be displayed. Each element in this list consists of two items, the signal value and the time at which the signal occurred. It is only necessary for the list to contain signal transitions; the intervening gaps will be filled in accordingly. For example, a signal which is initially unknown, but then falls low after three time units, and then rises high after four more time units is represented by the list {{0 X} {3 0} {7 1}}. If the specified signal is already present on the signal display, then its waveform will be modified according to the signals in the list.

The `sigdisp.tcl` module also contains two procedures which can be used by client code to delete signals in the signal display window — `sig_delete`, which accepts the name of the signal to delete and `sig_delete_all` which has no parameters. The implementation calls `sig_delete` whenever the user deletes a netlist label or a wire point which had a netlist label attached to it. The signal representing the netlist is removed from the display and the window is resized if necessary. The `sig_delete_all` procedure is invoked when the user wishes to delete the current circuit and start a new one. All the signals in the display will be removed, thereby restoring the signal display to its original state.

Two other procedures, `sig_rename` and `sig_type` affect the labelling of the signals. The `sig_rename` procedure accepts the original signal name and its new name and will change the name of the signal in the signal display. This procedure is called when the user changes the name of a netlist label via the Netlist Label dialog box. The `sig_type` procedure accepts the name of a signal and an optional parameter indicating the new type of the signal — either `input`, `output` or `unknown`. If the latter parameter is not supplied, then the procedure simply returns the current type of the specified signal. Otherwise, the type of the signal is changed to the specified type. This procedure is called when the user makes an adjustment to a netlist which results in a change to the existing netlist/port association. For example, if the user solders an output port to a labelled netlist, then the type of the signal representing the netlist must be changed to `output` in the signal display window. By doing this, signal types are automatically updated in the signal display as the netlist connectivity changes.

The `simulate.tcl` module is more closely related to the integration of the simulator engine with the GUI. As a result, the description of this module will be deferred until Chapter 5.

3.9 Miscellaneous Modules

Throughout the course of the implementation, several small utility procedures were written which were not directly related to the aforementioned modules. These modules consist primarily of procedures that were written to take care of repetitive tasks and to reduce code duplication. These miscellaneous modules and their descriptions are presented in Table 3.13.

Table 3.13: Miscellaneous Module Responsibilities

Module Name	Purpose
`fileslct.tcl`	Contains procedures for creating a file selection dialog box.
`mousewin.tcl`	This module will contain procedures that display help messages regarding the actions that will be carried out by each of the three mouse buttons in a particular context. This module is not yet currently implemented.
`optiondb.tcl`	Contains the single procedure `startup_init` which initializes the configuration option database. Default attributes for several widgets are established by this procedure.
`progress.tcl`	Create and update a progress bar. The progress bar is used during time consuming operations — as the operation nears completion, the bar grows longer.
`utils.tcl`	General purpose utility procedures used by the GUI.

Several of the procedures in these modules serve to extend existing Tcl/Tk functionality while others implement so-called "mega-widgets" which augment the existing widget set provided by Tk. Indeed, two of these modules, `fileslct.tcl` and `progress.tcl`, are generic enough to be integrated into applications completely unrelated to this project.

3.10 Simulation

This chapter has described the implementation of several aspects of the circuit editor GUI. One important note to make at this point is that the GUI maintains and validates structural information only — it is completely unaware of the behaviour of any of the components it creates. Therefore, apart from the different graphical representation, a 2-input *NAND* gate is treated exactly the same by the GUI as a 2-input *XOR* gate despite the very different behaviours exhibited by these two components.

In order to "give life" to a circuit, the GUI must be interfaced with a simulator engine which can take the structural information and the input signals provided by the GUI, simulate the circuit and then return resultant output waveforms to the GUI for display. The simulator engine itself is the topic of the following chapter.

Chapter 4

Simulator Engine

Chapter 1 presented an overview of simulation with special emphasis on the simulation of computer hardware. Numerous techniques for the description and simulation of hardware have been devised and implemented. Quite often, the technique adopted depends largely upon the desired level of abstraction at which the simulator engine is to operate. Some simulators accept low-level hardware descriptions and produce output which is based upon well known mathematical principles that govern the behaviour of circuit elements such as transistors and capacitors. At the other end of the spectrum, high-level simulators process hardware descriptions represented by functional or behavioural abstractions and generate simulation results which represent the behaviour of the system at this higher level. Typically, due to the often radical differences that exist between different levels of hardware abstraction, many simulator engines operate at only one level of abstraction. Other simulator engines are more flexible and attempt to accommodate descriptions at a few adjacent abstraction levels.

This chapter will focus upon describing and simulating hardware at the digital or gate-level using a discrete representation of time. What separates this simulator engine from other conventional digital simulators is the concept of *local time*. Each hardware component resides in its own temporal domain and is affected only by adjacent components. As will be shown, this concept makes it possible to both describe and simulate hardware hierarchically. A comparison of this approach with traditional digital simulator designs will be discussed. Another important theme emphasized by this chapter is the adoption of the *object-oriented* programming paradigm to design and implement the simulator engine. Adherence to the principles of this paradigm significantly facilitated the design of the simulator engine and resulted in an implementation which is easy to maintain and extend.

4.1 Simulation Using a Global Event Queue

The traditional approach towards gate-level simulation employs a global event queue which communicates directly with every component that participates in the simulation [9][12]. The queue is typically implemented as a linked list where each node represents a moment in time. Each node of the event queue contains a pointer to a chain of *events* which have been scheduled by the simulator to occur at that particular time. These events often consist of a pointer to the component which is to be simulated and an indication of what action the component must take.[1]

As events are processed and components are simulated, subsequent events are scheduled further ahead in time in the event queue. This scheduling of events represents how signals are propagated within the circuit — the global event queue is

[1]This action usually takes the form of the component being required to process a change in an input signal or to make a change in its internal state.

responsible for caching the outputs of a component and then feeding it to the corresponding fan-out components when the appropriate time node of the event queue has been reached during the simulation. The simulation runs until the event queue has been exhausted or until the time alloted for the simulation has expired. Figure 4.1 shows the relationship between a global event queue and the circuit that it is simulating.

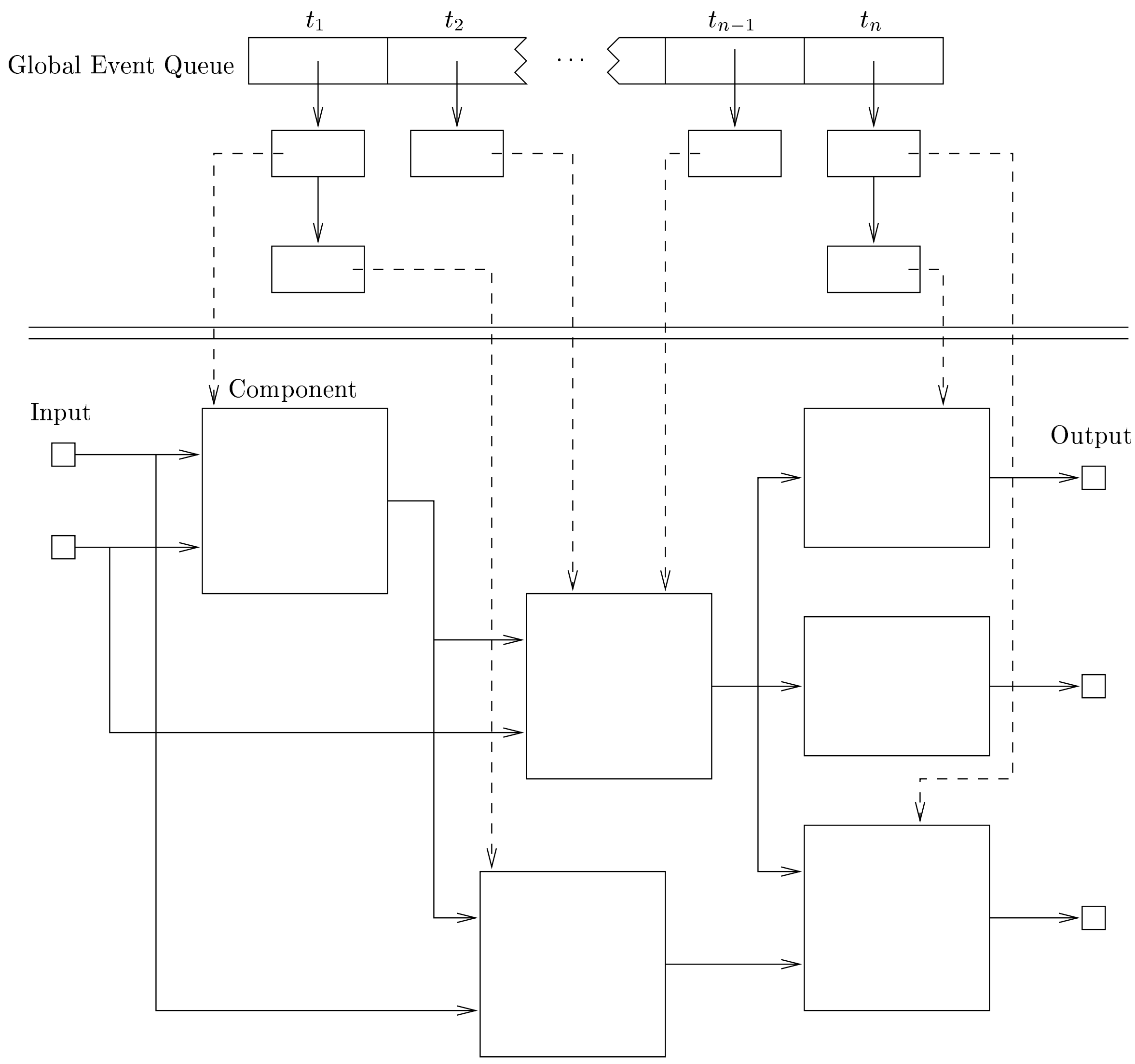
As can be seen from the figure, this method of simulation partitions the problem into two separate entities. The global event queue maintains all the simulation information; the circuit representation contains all the connectivity details. It will be demonstrated in subsequent sections how these two separate entities can be effectively merged into a distributed structure which maps naturally onto the wires of the circuit domain.

4.1.1 Drawbacks of the Global Event Queue

Despite the relative success of this simulation strategy, it still has several shortcomings, especially in the context of large scale simulations. Some of the problems are related to the separation of the circuit representation from the simulation while other problems manifest themselves as a result of the global nature of the event queue. Later sections will describe how replacing the global queue with a series of distributed queues can help to alleviate several of these problems.

A global event queue has no hardware equivalent. In the context of the implementation, the global event queue is an artifact which has no counterpart in an actual circuit. As a result, the implementation must bend the concept of a circuit so as to

Figure 4.1: Digital Simulation Using a Global Event Queue



accommodate this technique for hardware simulation. While this argument may be dismissed as a purely philosophical issue, it does have merit, especially when considering the translation of an abstraction into an implementation. The fewer artifacts an implementation introduces in mimicking the abstraction, the easier it becomes to understand and appreciate the implementation.

Two goals of software engineering and design are to decrease the coupling between unrelated modules while at the same time increasing the cohesiveness within functionally equivalent modules. Many purists may argue that the two tier approach towards digital simulation design serves to decrease the coupling between the simulator and the circuit representation thereby making the software more resilient to change. This assumption, however, is based upon the premise that the circuit representation and its simulation are two disparate entities. However, this separation is purely artificial. Instead, it could legitimately be argued that a design which combines the simulator and the circuit representation *increases the cohesiveness* of the implementation, since the two concepts are intimately related.

The importance of hierarchical representations of an abstraction was emphasized in Chapter 1 as a means of combating complexity. Hierarchical representation and subsequent simulation of circuits using a global event queue is potentially very convoluted as a result of the pervasive nature of the queue. Because each event in the queue contains a reference to the component to be simulated, it becomes very difficult to represent circuits hierarchically without compromising the autonomy of the low-level subcomponents of a component. One possible workaround to this problem is to place event queues at each level of the hierarchy and to devise a strategy whereby all the queues can be synchronized with one another via a top-level event queue. This may involve changing the concept of an event at the uppermost level of the simulation; instead of containing references to components and signals, events would instead contain references to other event queues. Needless to say, this concept of a global meta-event queue serves to only compound the problem of complexity rather than resolve it.

Related to the hierarchical representation and simulation of the circuit is the potential need to distribute large scale simulations over several different processes or even over several processors. Due to the global nature of the event queue, distribution of such a simulator would be very difficult. This may require the introduction of a process or a machine which is dedicated towards the synchronization of the event queues at each distribution point. Unfortunately, this introduces several problems. For example, if the synchronizing process was running on a separate machine, then that machine would undoubtedly act as a bottleneck during the course of the simulation since every other distribution node partaking in the simulation would have to communicate with it. Worse, if the machine performing the global synchronization went down, the entire simulation would be compromised.

Part of the blame for the proliferation of simulators which rely upon global event queues can be traced back to the languages with which they were implemented. Such languages tend to be structural in nature and provide only primitive support for true data abstraction and no support for inheritance or run-time binding. As a result of the relatively weak encapsulation support, the temptation to introduce global entities into an implementation is very strong. Subsequent sections will demonstrate how circuits may be simulated without the need for a global event queue by encapsulating some of the necessary simulation information within the circuit entities themselves. All the elements required for the simulation map directly onto analogous entities which exist in the real world. Central to the implementation described by this chapter will be the adherence to the principles of good object-oriented analysis and design.

4.2 Object-Oriented Approach Towards Simulation

With all the excitement generated over the past several years regarding the potential panacea derived from the creation of reusable software modules, several attempts have been made to modify existing algorithms and implementations so that they adopt a decidedly more object-oriented flavour [13][16][30]. Digital circuit simulation has proven to be no exception to this trend. Unfortunately, as will be demonstrated by this section, such attempts tend to be somewhat naïve in their approach and sometimes amount to nothing more than a translation of the global event queue to the realm of objects.

4.2.1 Examples of Digital Simulator Designs

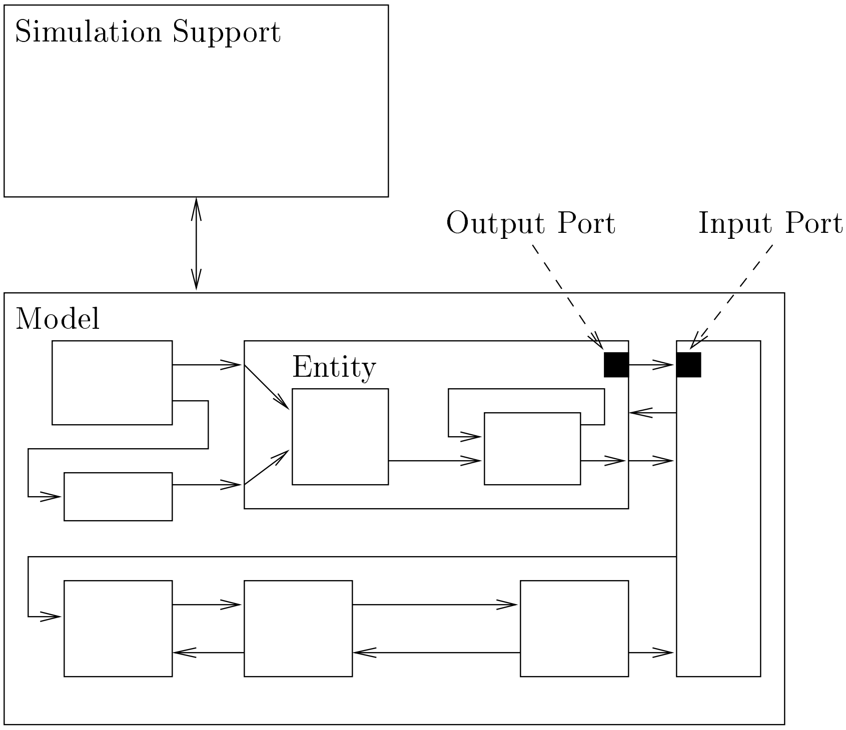
Many of the existing simulators adopt very similar approaches towards the classification of circuit elements, as will be discussed in a subsequent section. Where many of the simulator designs start to deviate, however, is in the area of component interconnectivity and simulation. Many of the implementations still enforce the need for a separation between the representation of the circuit and its ensuing simulation.

For example, *DOSE* [16] provides the anticipated classes for components, input ports and output ports upon which a discrete event-driven simulator may be based. It also supports the notion of composite components for the purposes of hierarchical modelling. However, the class library also introduces an independent simulator object which is composed of three other objects; namely, a component manager, connection manager and event scheduler. The interaction between these three objects essentially provides the necessary support for event-driven simulation. One major drawback to this technique is that the pervasive nature of these three objects tends to make the components less autonomous and more dependent upon global entities to provide them with inputs and to propagate their outputs. This dependence can seriously compromise the ability to distribute the simulation and can make hierarchical modelling unwieldy.

A second approach [13] is somewhat similar to above, yet provides better support for distributed simulation. A system overview of its architecture is presented in Figure 4.2. As can be seen from the figure, the architecture creates a clear separation between the simulation support subsystem and a model representing the entity being simulated. This simulator implements a sophisticated hierarchical event handling mechanism. If an entity receives an event it is unable to handle, the event is propagated to its parent. However, the concept of a "global simulation control manager" is still required for the purpose of phase change notifications. In addition, the system supports the concept of a *model entity* class which contains event lists. These models may be nested, thereby giving rise to the concept of multiple event lists. The literature is careful to point out though, that "[if] event lists are distributed, the models are responsible for synchronizing them." Consequently, the implementation still perpetuates the concept of global time and emphasizes the necessity for synchronization when the simulation is distributed over several machines.

Subsequent sections will discuss the design and implementation of a simulator engine which is largely asynchronous in nature. Concepts such as global time and global event queues will be abandoned in favour of local time and distributed event queues.

Figure 4.2: System Overview of a Hierarchical Simulator



4.3 An Alternative Approach Towards Simulation

This section describes a different and possibly more intuitive approach towards the design and implementation of a digital simulator. This implementation builds upon the simulator engine described in [7]. While the simulator engine works primarily at the gate-level of abstraction, it should be noted that the engine can conceivably be adapted to operate at higher levels.[2]

In general, the heart of the simulator engine is comprised of a distributed queuing mechanism in which each component maintains its own local time. In this manner, all the components are relatively autonomous and are influenced only by adjacent

[2]Due to the inherent discrete representation of time, simulation at the lowest circuit-level, which requires a continuous representation of time, is not immediately realizable by this simulator engine.

incoming queues. This technique provides a means by which the simulation of a circuit may be integrated with its representation, thereby eliminating the artificial distinction between the two. The concepts behind local time and distributed event queues are elaborated upon in the following subsections.

4.3.1 The Concept of Local Time

The notion of local time is contrary to the aforementioned traditional approaches in which all components are kept at the same point along the time line during the entire course of the simulation. Synchronization of all the components in such a simulator is enforced by a global queue that must invade the autonomy of all the components being simulated. By assigning the components the ability to manage their own local time, the components are not subject to the whims of an overburdened global event queue. As will be shown later, the concept of local time makes it possible to cleanly represent and simulate a circuit hierarchically.

The idea of local time is also exploited by the Chandy-Misra algorithm [5] as a means of achieving asynchronous distributed simulation in a parallel environment. Their motivation for employing a local time mechanism is justified by their need to eliminate the potential bottleneck introduced by a global entity which synchronizes all the simulation components:

> "We do not wish to use any global variables nor do we want to use a single process to drive the simulation because it will prove to be a bottleneck. Our approach is totally asynchronous; every process maintains its own local clock and there is no global synchronization mechanism such as a global clock." [5]

In some respects, the simulation algorithm described by this section is similar to the Chandy-Misra algorithm; however, the algorithm has been adapted to run in a uniprocessor, sequential instruction environment.

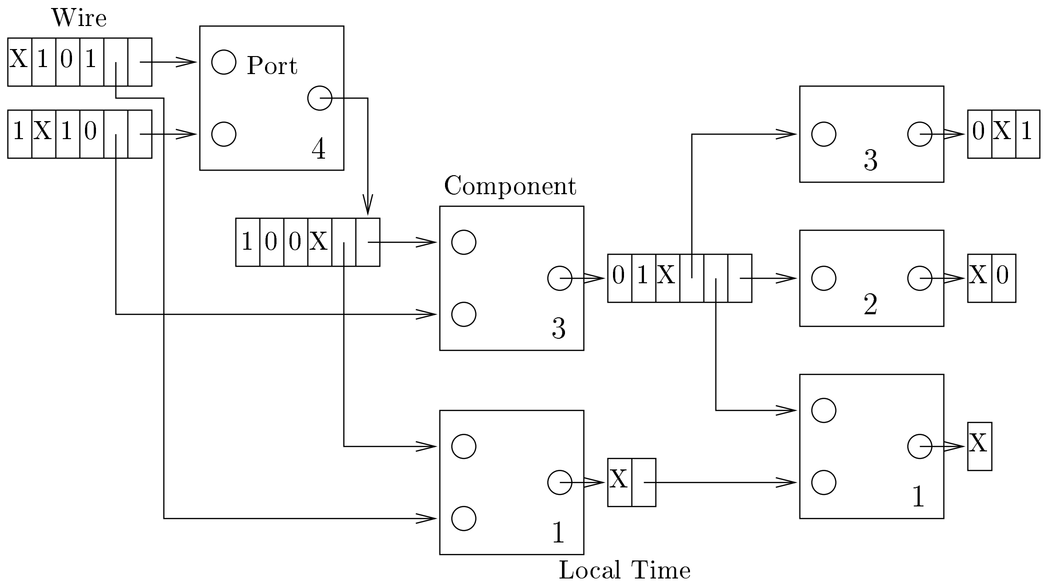
One of the ramifications of the concept of local time is that during the course of the simulation, some components will be further advanced along the time line than other components. Indeed, it is even possible that some components may complete their contribution to the simulation before other components have even received any input at all. This fact creates an opportunity for memory recovery during the course of the simulation. For example, consider the case where a relatively large subcomponent has consumed all of its inputs and has successfully generated all of its outputs. At this point, the subcomponent is only wasting memory since it can no longer affect the outcome of the simulation, hence it can safely be destroyed. New subcomponents can subsequently be created and attached to the circuit. Through careful implementation, it may be possible to take advantage of this fact to simulate circuits which are too large to fit into memory all at once. Although the current implementation of the simulator engine does not support this strategy, it still seems theoretically plausible.

4.3.2 Distributed Event Queues

Local time contributes to the preservation of the autonomy of components during a simulation. However, in any simulation, components still require some means of communication with other components. Ideally, the method of interaction between components should be intuitive and should not introduce bottlenecks into the simulation by relying upon the presence of a global mechanism. For this reason, distributed event queues were introduced to help maintain the autonomy of a component by encouraging interactions amongst adjacent components only. As with local time, distributed event queues are also amenable to the hierarchical representations of circuits.

Figure 4.3 shows a simulation in progress which employs distributed event queues. The circuit is identical to that presented in Figure 4.1 but the global event queue has been eliminated. Instead, the event queue has essentially been distributed throughout the circuit representation. These distributed queues are analogous to wires in actual circuits, in that they serve as the conduits in which signals pass from one component to other components in its fan-out thereby connecting components together.[3]

Figure 4.3: Digital Simulation Using Distributed Event Queues



In addition, these distributed queues also maintain a history of the events (or signals) which have travelled along them during the course of the simulation. Hence, when a component is being simulated, it may query its input wires (via its ports) to obtain the value of the signal which occurred at local time of the component. If the input signals are not available in the queue at the requested time, then the component

[3]A component may also be connected to itself, therefore allowing feedback loops to be simulated.

will not be permitted to simulate. As demonstrated by the figure, it is necessary for the event queues to maintain a history of their signal values since the local times of the components to which they are attached may vary significantly.

The following subsections describe how distributed event queues serve the dual function of expressing connectivity and facilitating the simulation of a circuit. Details with respect to circuit representation and class design strategies will be discussed in depth.

4.3.3 Circuit Classification and Representation

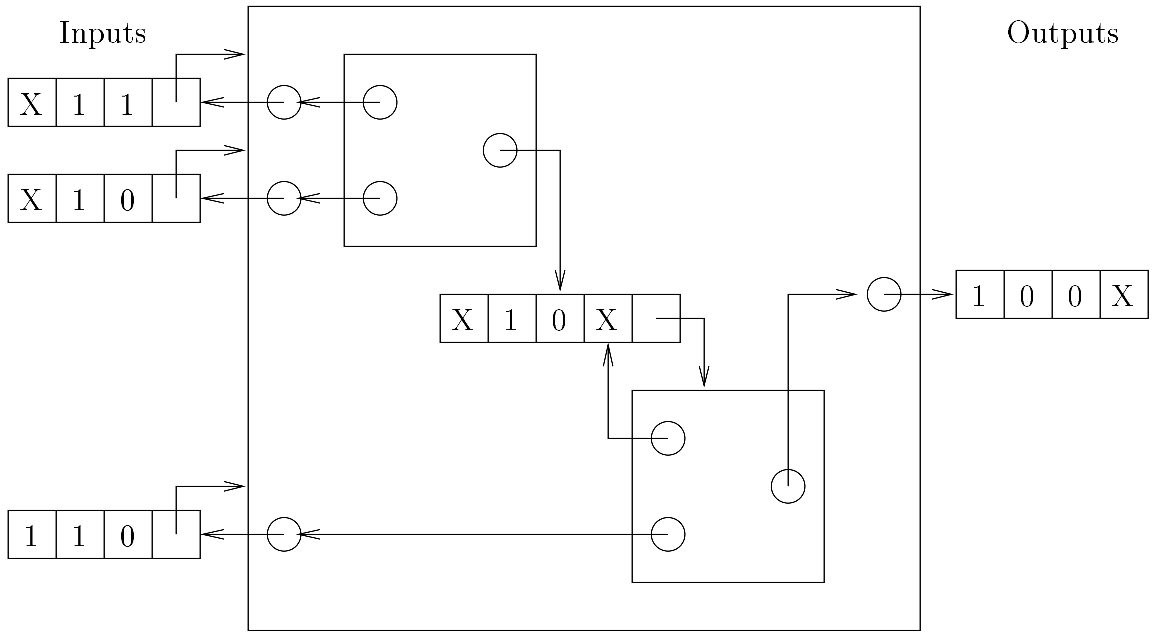
Circuit representation presents many challenges for the designers of simulator software; the potential complexity of circuits only serves to compound this challenge. This section provides the necessary infrastructure and insight upon which our subsequent circuit class design will be based.

One of the major goals of our circuit representation is to enable the expression of the hierarchical nature of circuits so as to help offset the inherent complexity associated with circuit design. By allowing components to be represented as a composite of subcomponents, it becomes easier to construct higher level components by connecting together rudimentary components whose behaviours are more easily understood. These high level components may then become subcomponents of even larger, more powerful circuits. Hierarchical representations also present the possibility for distributing the circuit and its subsequent simulation over several machines. A second goal of our representation is to model real world circuits as closely as possible by limiting the number of artifacts introduced into the representation. By modelling circuits as closely as possible to their real world counterparts, we must also consider incorporating support for simulation into our representation. As will be discussed later, the distributed queues facilitate this objective.

The process of "*discovering* the classes and objects that form the vocabulary of the problem domain" constitutes the *analysis* phase of the object-oriented paradigm [3]. Several similar classification schemes have been adopted by many object-oriented approaches with respect to circuit representation. Generally speaking, object-oriented implementations classify circuits in accordance with real world circuit entities such as components, ports and netlists. This classification strategy effectively creates a mapping of software classes onto analogous elements in the domain of real world circuits, thereby leading to a better understanding of the resulting implementation by others.

With respect to circuit representation, the concept of creating a base component from which all other functioning components can be derived seems to be a common classification strategy throughout the literature. Aspects shared by all components are factored out and placed into a base class. All subsequent components and other functioning units are then derived from this base class, thereby inheriting the commonality. Most implementations also provide support for nesting components within one another, thereby permitting hierarchical decomposition of circuit representations. A base class representing ports is also popular; from this class, an input and output port class may then be derived. Because this approach to classification is relatively natural and, indeed, almost intuitive, the simulator engine that forms the focus of this report adheres to a similar classification approach with respect to circuit representation.

Figure 4.4: Representation of a Simple Composite Circuit



The representation of a circuit will be discussed in the context of a specific example. Figure 4.4 presents an elementary circuit which has three input ports and one output port. This circuit is composed of two nested components, each of which have two inputs and one output; and a wire which connects these two subcomponents together. The primary classes emerging from this example are components, ports and wires. The intended roles of these classes are as follows:

- **Component**: A component is the functional block of the circuit. It may exist as an independent fundamental unit or as a higher level unit which indirectly delegates functional responsibilities to its subcomponents.

- **Port**: Ports connect components vertically to adjacent hierarchies and horizontally to adjacent wires. Ports can be thought of as connecting components to the "external world."

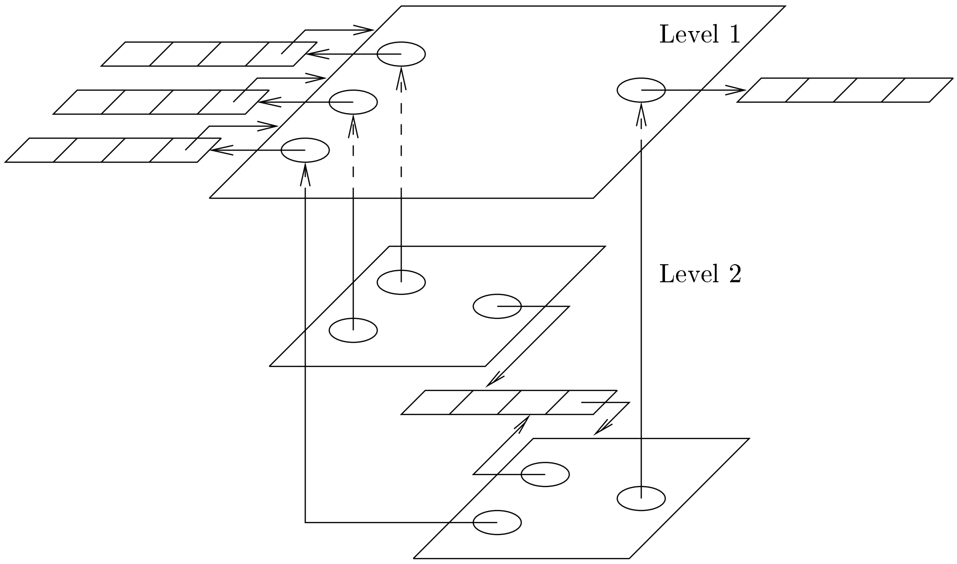
- Wire: From a purely representational viewpoint, wires serve to connect together components at the same hierarchical level via the components' ports. From a simulation perspective, they are the distributed queues and act as "event archivers."

In order to more vividly convey the hierarchical potential of this representation, consider the three-dimensional hierarchy presented in Figure 4.5. This circuit is identical to the one presented in Figure 4.4, except that it is tilted slightly so as to make the distinction between hierarchical levels more obvious. Of particular interest is the mechanism by which components lower in the hierarchy communicate with entities higher in the hierarchy. In the figure, for instance, note that the ports of the lower level components do not communicate directly with the wires at the top level. Instead, the ports of the subcomponents interface with the ports of the top level component, which, in turn, communicate with the wires at the top level. This mechanism helps to preserve the encapsulative nature of the top level component and all its subcomponents.

4.3.4 Class Design

During the design of a model, "we *invent* the abstractions and mechanisms that provide the behaviour that this model requires." [3]. Design typically represents the stage in the software process just before the implementation phase; the quality of the design quite often determines the extensibility, robustness and maintainability of the ensuing implementation. Because a bad design can seriously compromise the

Figure 4.5: Three-dimensional Hierarchical Circuit Representation



integrity of the subsequent implementation, it is important that the design phase not be treated lightly.

To aid in the discussion of the design of the simulator engine, several *Booch diagrams* [3] will be presented which highlight many of the important classes of the simulator architecture and the relationships among them. Booch diagrams illustrate the major aspects of a design through the use of several well-defined icons and adornments. In a class diagram, the most important icon is the "dotted cloud" which symbolizes a class; lines connecting two class icons together suggest an association between the two classes. By applying various adornments, such as arrows, circles and squares, to the end points of an association line, the designer may specify the type of association that exists between the two classes.

With respect to object-oriented design, there are two fundamental relationships which will be of particular interest to us, namely, the *is a kind of* relationship and the *is a part of* relationship. Two classes which exhibit the *is a kind of* association are potential candidates for inheritance. For example, a dog *is a kind of* mammal, therefore it would be natural for a Dog class to inherit from a Mammal class; thereby causing the Dog class to acquire all the attributes of the Mammal class. This type of relationship is also commonly called *specialization/generalization*. The *is a part of* relationship, however, suggests that an aggregation association exists between two classes, whereby one instance of a class can contain an instance of another class. For example, biologically, a spleen *is a part of* a dog; consequently, it is acceptable to encapsulate a Spleen class instance inside an instance of the Dog class so as to express this relationship. Aggregation associations are also referred to as *whole/part* relationships. The following subsections will make extensive use of these relationships, and others, when describing the various associations between classes in the domain of circuit representation and simulation.

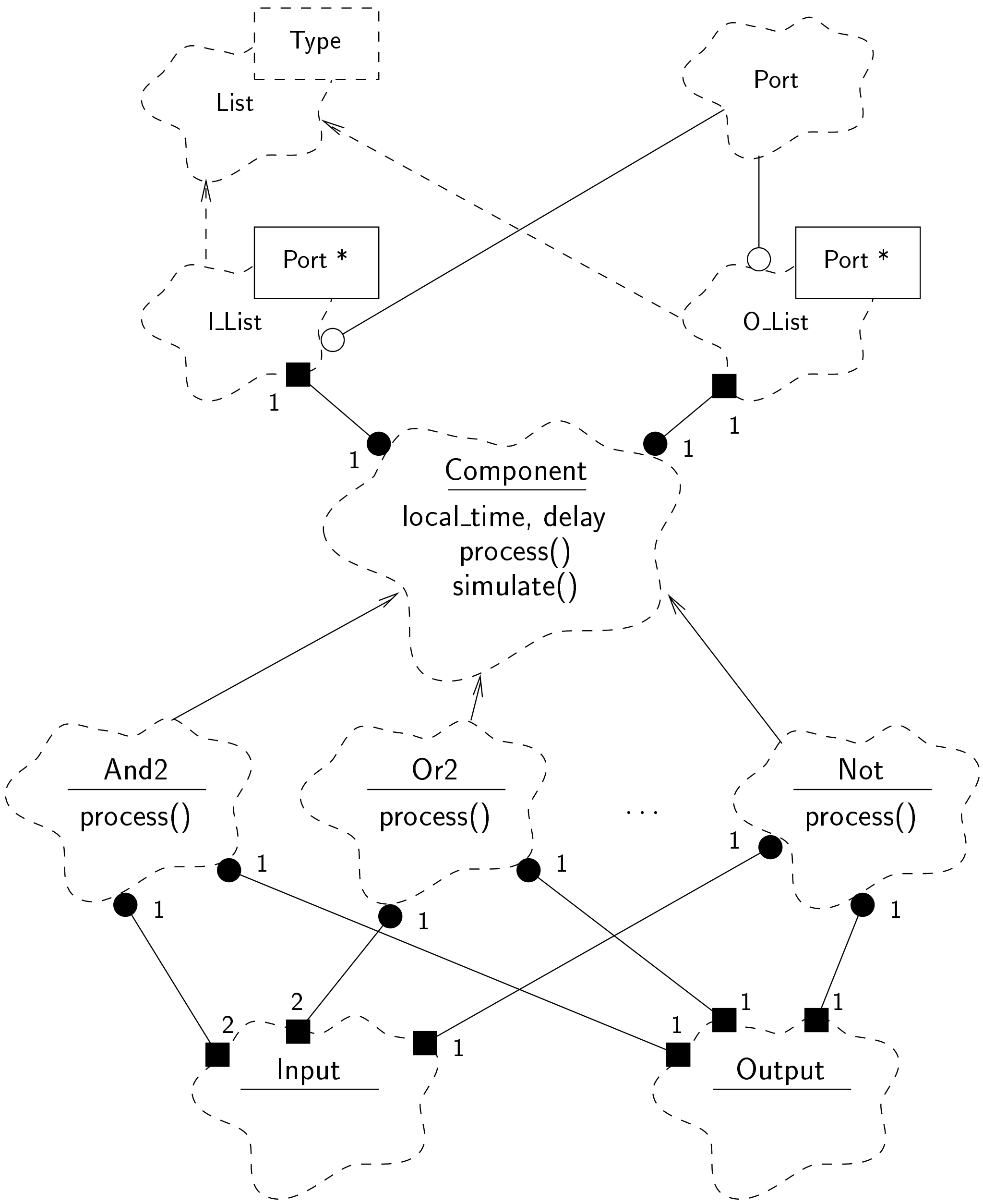
The Component Class

Figure 4.6 shows the Booch diagram for the Component class and many of its related classes. The Component class itself is represented by the dotted cloud icon in the center of the diagram. The icon shows that this class has two data members (`local_time` and `delay`) and two member functions (`simulate()` and `process()`).[4] The `local_time` data member contains the current local time of the class and the `delay` data member contains the transport delay of the component, which is initialized when the component is constructed. The `simulate()` and `process()` member functions are primarily responsible for the simulation aspects of the component; these

[4]Actually, the Component class contains more than these data members and member functions. However, only the data members and methods which are of particular interest are usually displayed in a Booch diagram.

details will be deferred until Section 4.3.5.

Figure 4.6: Component Class Diagram



Next, looking above the Component class, there are two adjacent classes named I_List and O_List. They are both associated with the Component class by a line with a filled circle at the Component end and a filled rectangle at the I_List/O_List ends. These two classes represent the list of input ports and output ports for the component. The filled circle symbolizes a containment or aggregation relationship. Because the input and output port lists are *part of* a component, this relationship is justified. The filled rectangle implies that the aggregation is a *physical* containment as opposed to a *pointer/reference* containment. By using physical containment, we are ensured that the lifetimes of the input and output port lists during the simulation will be the same as the lifetime of the enclosing component. The two small numbers adjacent to the end points of each of the lines signify the cardinality of the relationship. In other words, each component contains one input port list and one output port list.

Note that the I_List and O_List class icons each have a solid rectangle in their respective upper right regions. This adornment indicates that these two classes are actually instances of a parameterized class.[5] The dotted lines with the arrow heads emanating from the I_List and O_List classes indicate that these two classes were instantiated from the List parameterized classes. This parameterized class provides rudimentary support for a generic linked list structure which can be manipulated in a type-safe manner. The text inside the solid rectangles of the I_List and O_List classes represent the actual arguments to the List parameterized class; the text inside the dotted rectangle of the List class represents the formal arguments. As with parameter passing in procedural languages, an association is established between the formal arguments and the actual arguments. In the context of this specific example, a correspondence between the Port * actual argument and the Type formal argument is

[5]In our implementation, parameterized classes are implemented using the C++ *template* mechanism.

created, thereby transforming I_List and O_List into list classes which contain pointers to Port objects. Note that in order for the instantiation to occur, the I_List and O_List classes both require the services of the Port class. This *using* relationship is shown by the solid line emanated from the Port class and ending with a hollow circle on the I_List/O_List classes.

Finally, we focus on the classes immediately below the Component class in the figure. These classes represent some of the lowest-level units of the component library, such as 2-input And and Or gates and a 1-input Not gate. They are related to the Component class by a solid line with an arrow pointing towards the Component class. This is an *inheritance* relationship — an And gate *is a kind of* component, hence, we can derive And from Component. Note that each of the low-level gates contain their own `process()` method, thereby overriding the virtual `process()` member function in the base Component class.

The gates are also related to the Input and Output classes via several aggregation relationships, similar to the ones mentioned earlier. These relationships are used to illustrate the number of input and output ports contained within each component. For example, a 2-input Or gate contains two inputs and one output. Therefore, as the cardinality in the diagram shows, the Or gate physically contains two objects instantiated from the Input class and one object instantiated from the Output class.

On the surface, it may seem as though each component maintains two different sets of input ports and two different sets of output ports — one such set is inherited from its base Component class, and another set is created from its aggregate data members when it is constructed. However, this is not true. Remember that the Component contains two lists which store *pointers* to Port objects and *not* Port objects themselves.

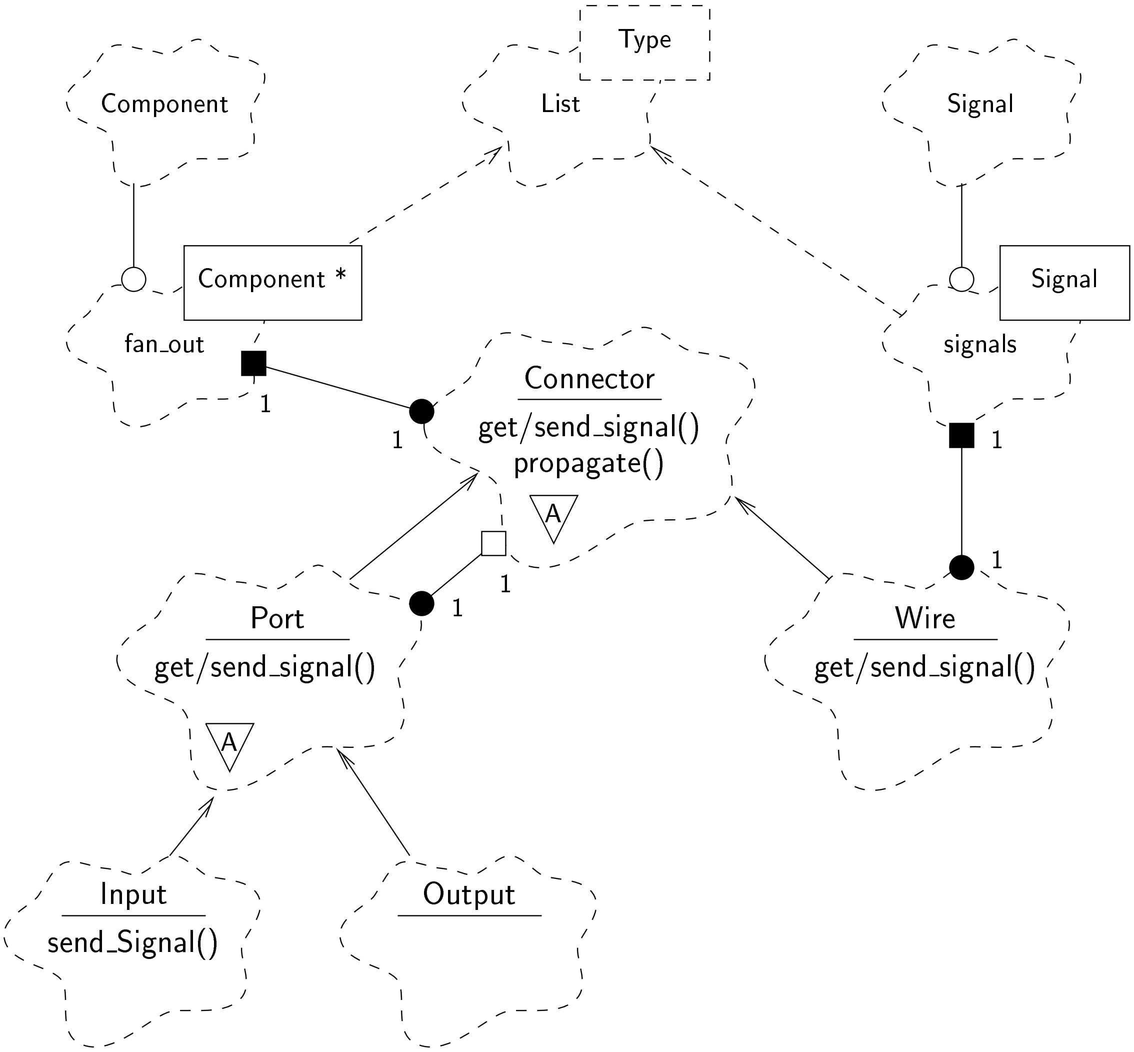
The Port objects are actually created by classes derived from the Component class. As such, when a derived component is instantiated, its constructor will add pointers to its input/output ports to the corresponding port lists it inherits from the base Component class. In this way, the base Component class can have a generic mechanism which can traverse input and output ports, even though it doesn't know in advance how many such ports its derived components will have. This mechanism is then inherited by all of its derived classes instead of having to be duplicated in each one.

The Connector Class

Now that we have provided a means by which functional units of the circuit may be created, we next must devise a strategy by which these units may communicate with one another and with their external inputs and outputs. As alluded to earlier, communication between components will be handled by the Port and Wire classes. However, instead of simply creating these classes independently from one another, we note that both these classes have a common underlying theme; namely, they are responsible for *connecting* entities together. Hence, it is natural to think of ports and wires as being *kind of* connectors, since ports connect components to the external world and wires connect two or more components together. Therefore, we create a common base class called Connector from which we subsequently derive the Port and Wire classes. These derivations are illustrated in Figure 4.7.

The Connector class contains two virtual member functions, `get_signal()` and `send_signal()`. These methods are responsible for obtaining and transmitting signals respectively, hence providing the necessary facilities for inter- and intra-component signal flow. Note that the Connector class itself does not actually define these methods.

Figure 4.7: Connector Class Diagram



Instead, it relies upon the specialized classes derived from it to actually implement the semantics of the two methods. Virtual functions which cannot be implemented by a base class due to its high level of abstraction are commonly referred to as *pure virtual functions*. Because the Connector class contains pure virtual functions, no objects can be instantiated from this class. A class which cannot have instances is called an *abstract base class*.[6] As can be seen from the diagram, abstract base class

[6]Note that even though objects may not be instantiated from an abstract base class, it is perfectly

icons are adorned with the letter A enclosed in an inverted triangle. Objects may be instantiated from classes derived from an abstract base class provided that the derived class defines all the pure virtual functions of its base class.

To the left of Connector in the figure is the fan_out class. Because connectors are responsible for connecting components together, they must maintain a list of all the components to which they are connected for the purposes of signal propagation and vertical hierarchy traversal. The fan_out class is associated with the Connector class by a physical aggregation relationship — each Connector class physically contains a list of pointers to components. The fan_out class acquires its list handling capabilities by instantiating from the List parameterized class and by using the Component class. The `propagate()` member function of the Connector class traverses the components in the fan-out list and sends Component::`simulate()` messages to each of them.

The Wire class, as mentioned earlier, is derived from the Connector class. It physically contains a signals list which is instantiated from the List parameterized class using the Signal class. The Wire class overrides the `get_signal()` and `send_signal()` methods. The `get_signal()` method simply traverses the wire's linked list of time ordered signals searching for a signal which occurred at a specified time and returns the signal. The `send_signal()` method of the Wire class adds a specified signal to its linked list of signals and then attempts to propagate it to all components in its fan-out component list using the `propagate()` method it inherited from its base Connector class.

The Port class is derived from the Connector class. This class overrides both

valid and indeed necessary in many circumstances, to create pointers to these base classes. The fact that a derived class pointer can be assigned to a pointer to its public base class without a cast forms the foundation upon which polymorphism is based.

the get_signal() and send_signal() methods of its base class. In addition to the derivation relationship between the Port class and the Connector class, an aggregation relationship also exists between these two classes. Unlike previous containment associations, the filled rectangle has been replaced with a hollow rectangle. This adornment suggests containment by pointer (or reference) — the Port class contains a pointer to a Connector class. This pointer is referred to as the Port's *external* connector. Because a pointer to a base class can legitimately point to any of its derived classes, this external Connector pointer can point to either another Port class or to a Wire class.

This pointer containment relationship is required for the hierarchical transmission of signals throughout the circuit. Both the get_signal() and send_signal() methods of the Port class exploit this relationship so as to vertically traverse the hierarchical representation during simulation. The get_signal() method of the Port class travels recursively up the hierarchy via the external connectors until it encounters a wire, at which point the get_signal() method of the Wire class returns the requested signal from its signal list. Similarly, the send_signal() method of the Port class transmits a signal out via a series of external connectors until a wire is encountered, at which point the send_signal() method of the Wire object stores the signal and propagates it. The send_signal() method of the Port class then propagates the signal to all components in its fan-out. The two send_signal() methods of the Port and Wire classes are presented in Figure 4.8 and Figure 4.9, respectively.

Note that the abstract class adornment is still present on the Port class. This is because the Port class is still too generic to be used for object instantiation. We prevent instantiation of the Port class by making its constructor protected, thereby

Figure 4.8: Sending a signal to a Port

```
void Port::send_signal(Signal sig)
{
    external->send_signal(sig);
    propagate();
}
```

Figure 4.9: Sending a signal to a Wire

```
void Wire::send_signal(Signal sig)
{
    add_signal(sig);
    propagate();
}
```

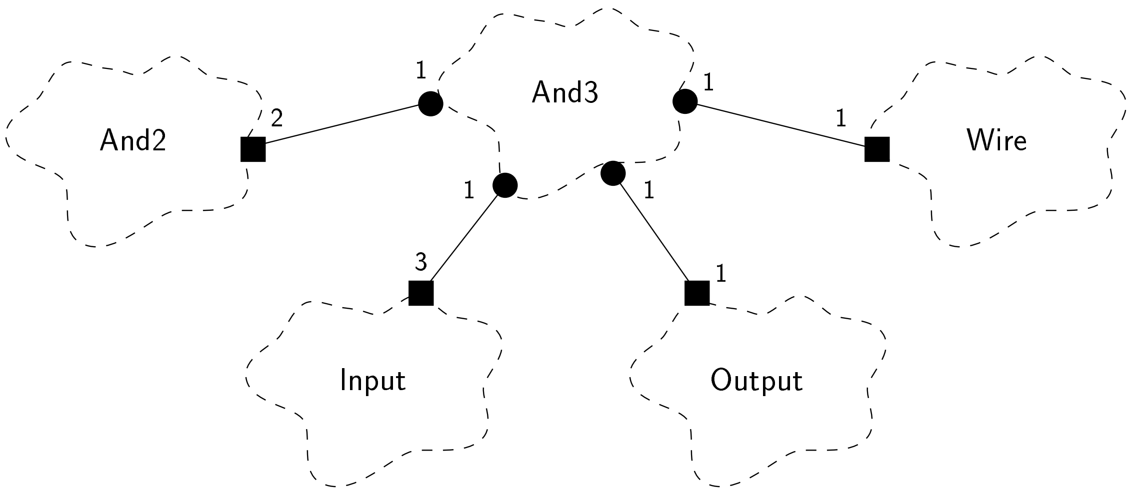
implying that only derived classes may call its constructor. These derived classes can subsequently be used for instantiation.

Finally, the Input and Output classes are derived from the Port class. Because the constructors for these two classes are public, components may include input and output port objects as part of their representation. The only major point of note here is that the Input class redefines its `send_signal()` message to display an error instead of sending a signal, because input ports can only receive signals, not send them.

By using various combinations of classes derived from the Component class and Connector class, it is possible to build arbitrary blocks of logic. For example, consider the construction of a 3-input *AND* gate using two 2-input *AND* gates. Such a gate would contain three input ports, one output port, two instances of 2-input *AND* gates and a wire. The constructor would then be responsible for connecting all the ports and wires together in some meaningful fashion. A Booch diagram representing a

3-input *AND* gate is presented in Figure 4.10.

Figure 4.10: Class diagram of a 3-input *AND* gate



The only issue left to address is how the classes described above interact to simulate a circuit. The fundamentals behind the simulation algorithm are discussed in the next section.

4.3.5 The Simulation Algorithm

In simplistic terms, the simulation algorithm amounts to nothing more than a depth first traversal of the aforementioned three-dimensional hierarchy of components, in which signals are propagated both horizontally across the same hierarchical level and vertically through different hierarchical levels. One of the primary differences between this approach and the classical event-driven approach is with respect to *when* events are propagated. In the global queue approach, events are entered into a sorted event list and are propagated later; whereas in the distributed queue approach, events are entered into an entity's output queue and are propagated immediately, if possible.

The algorithm itself is set in motion when the top-level component receives a `simulate()` message. The semantics of this message are fairly straightforward and can be summarized with the following pseudo-code:

Component::`simulate()` *`/*` Same for all components `*/`*

while (component inputs are ready at *local time*) **do**

increment *local time* of component

process inputs at *local time - 1*

done

Note that it is important that the local time of the component be incremented *before* the `process()` method is called. If this is not done, then the local time will *never* be incremented in cases where feedback is present in the circuit description. This would result in a non-terminating simulation.

Because the `process()` method is virtual, the actions taken by the second line of the while loop depend upon whether the component overrides the `process()` method it inherited from the Component base class. Components which are composed of sub-components do not typically provide their own `process()` method. Instead, they simply inherit the same method as defined in the Component base class. The behaviour of the `process()` method in this case is to traverse the input port list of the component and to send `simulate()` messages to all the embedded subcomponents, as demonstrated by the following pseudo-code:

```
Component::process(t)                    /* For high-level components */
    for each (input port of component) do
        for each (component in fan-out of port) do
            simulate component
        done
    done
```

Hence, the default behaviour of the `process()` method is to essentially descend the representation hierarchy, informing lower-level components to process their inputs. A component which contains no subcomponents and does not override the `process()` virtual function of the base `Component` class is effectively treated as a null component since it is not capable of producing output.[7]

Consequently, components which are located at the lowest level of the hierarchy (that is, components that do not contain subcomponents) should provide their own `process()` method, thus overriding the `process()` method in the base class. In this case, the `process()` method will typically employ the method `Port::get_signal()` to obtain the input signals that occurred at the current local time of the component. The `process()` method then performs some logic or calculation based upon these inputs and then sends the new signals to the output ports of the component using the method `Port::send_signal()`. An example of such a sequence of operations is

[7] A trivial modification to the simulator engine could detect null components during run-time and display a diagnostic error message when they are encountered during the simulation.

demonstrated by the following pseudo-code:

```
Component::process(t)                    /* For low-level components */
    get inputs from inputs ports at time t
    calculate outputs based upon inputs
    send outputs to output ports with timestamp t + delay
```

Note that when adding a new low-level component that requires its own `process()` method to an existing component library, there is no need to add a new `case` label to a lengthy `switch` statement to ensure that the correct `process()` function is invoked for the new type of component. The virtual function mechanism will invoke the correct `process()` method for the component at run-time.

The implementation of the simulator engine, is for the most part, independent of the GUI discussed in the previous chapter — the simulator engine may be run without the GUI and vice versa. This feature results in loose coupling, and therefore makes both the GUI and the engine reusable as separate autonomous modules in their own right. Details concerning the integration of the GUI and the simulator engine are presented in the next chapter.

Chapter 5

System Integration

Chapters 3 and 4 described the implementation of a GUI for digital circuit layout and a simulator engine for digital circuits, respectively. These two software entities can each operate as stand-alone applications; unfortunately, they are each limited in their functionality. The GUI layout application has no integrated simulation capabilities and the simulator engine has a very primitive interface by which users must tediously describe the circuits and their inputs textually and then decipher the textual signal outputs after simulation. This chapter describes how these two separate entities have been integrated together to form a single application from the perspective of the end user, thereby providing the benefits of both software units. The advantages of this highly modular approach towards system integration will also be discussed.

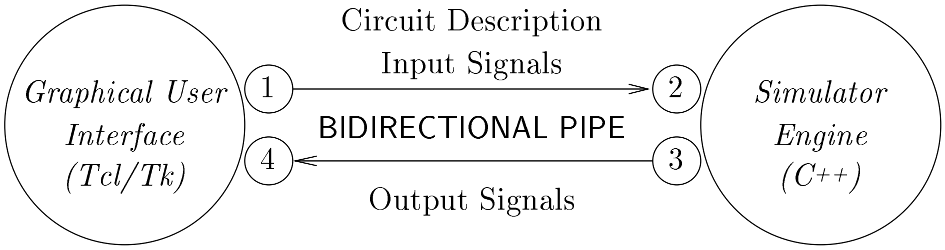
5.1 System Integration Techniques

In order to integrate the GUI and the simulator engine, we must devise a means whereby they may communicate with one another. In essence, we want the GUI to be able to send the input signals and circuit description to the simulator engine and we want the simulator engine to be able to transmit the resultant output signals back to the GUI for presentation in the signal waveform display window. As a result, the chosen communication mechanism must support bidirectionalism between the two software units. Adding to the difficulty of integration is the fact that the GUI was written entirely in Tcl/Tk while the simulator engine was implemented in C++. Hence, linking the two modules together in the conventional sense is not an option, since a relocatable object module cannot be generated for Tcl scripts.

One integration strategy would be to embed the Tcl script into a C program, devise a functional interface between the C program and the C++ simulator and then to link the resulting object modules together. However, this method was rejected because it is somewhat complicated and because it may contribute to obscuring the distinction between the layout editor and the simulator engine. As will be justified later, preserving the relative autonomy of the GUI and the simulator engine contributes significantly to the potential reusability of each unit.

After weighing some of the options, it was decided that a *command pipeline* should be used as the fundamental conduit of information transfer between the GUI and the simulator engine. Using a pipe, a simple protocol could be developed by which the GUI and simulator engine would communicate with one another. In addition, the only means by which the GUI and the simulator would interact with each other is via the pipe, thereby encouraging loose coupling between the two modules. A simple, high-level overview of the system model showing how the pipe enables interaction between the GUI and the simulator engine is presented in Figure 5.1. The numbers located in the small nodules of the figure represent the four stages of the interaction between the two modules. These stages are discussed in detail in Section 5.4.

Figure 5.1: Communication between the GUI and Simulator Engine



Of particular importance is the bidirectional nature of the command pipeline. Many software applications already employ pipes to achieve communication between two or more separate software modules. However, that communication is usually unidirectional — one software module produces output which is then manipulated and supplied as the input to another application over the pipe. At this point, the latter application cannot send information back through the pipe to the original module. This may severely limit the functionality and flexibility of the software application as a whole. A bidirectional pipe, however, makes it possible for two or more applications to operate as peers. In such a situation, all of the modules have the ability to send data to and receive data from one another. This makes it possible to achieve more flexible behaviour, especially on the context of several modules operating in parallel.

5.1.1 Integrating Modules Using Command Pipelines

Before discussing the specific details of how a command pipeline is used to enable the GUI and the simulator to communicate with one another, a brief overview of pipes and a small example of their operation is presented.

Consider, for example, a situation in which a Tcl script wishes to send two numbers to a subprocess which will add these two numbers and then send back the sum.[1] Such a Tcl script is presented in Figure 5.2. Because the first character of the first argument to the `open` command is a vertical bar, commonly referred to a *pipe* symbol, the `open` command will actually execute the program given by the text following the pipe character (`addnum`) as a subprocess. A command pipeline is opened to that process; the standard input and standard output of the subprocess are tied to the file identifier, `fid`, returned by the `open` command. Whenever the subprocess reads from standard input it is actually reading information put into the pipe by the Tcl script via the file identifier. Likewise, any information that the subprocess attempts to display on standard output will be intercepted by the Tcl script through the file identifier.

Figure 5.2: Tcl Script Opening a Pipe to an Executable.

```
#!/usr/local/bin/tclsh

set fid [open "|addnum" "r+"]   ;# Open pipe to executable.
puts $fid "18"; puts $fid "24"  ;# Send two numbers to add.
flush $fid                      ;# Flush the output buffer.
set result [gets $fid]          ;# Read back the sum.
close $fid                      ;# Close the pipe.
puts "The sum is $result"       ;# Output the sum.
```

The subprocess, `addnum`, invoked by the above Tcl script can be written in any language, either interpreted or compiled. The only assumption made by the Tcl script

[1]This example may not be as contrived as one might think. Since Tcl scripts are interpreted, it is naturally going to execute more slowly than a compiled binary. As a result, if one is evaluating a convoluted expression that involves numerous iterations and time consuming control flow, then sending the raw data down a pipe to an executable for processing and then reading the result back may actually be faster than performing the entire evaluation in native Tcl.

of the subprocess is that it take two numbers from standard input and produce the sum of these two numbers on standard output. Such a program, written in C++, is presented in Figure 5.3. Assuming that the compiled `addnum` binary is in the same directory as the Tcl script, the end user need only run the Tcl script; the `addnum` binary will be executed transparently to the user.

Figure 5.3: The C++ Program `addnum`.

```
#include    <iostream.h>

int
main()
{
    int num_1, num_2, result;

    cin >> num_1; cin >> num_2; // Read the two numbers.
    result = num_1 + num_2;     // Evaluate result.
    cout << result;             // Print the result.
    return 0;                   // Execution successful.
}
```

5.2 Advantages of Using Command Pipelines

Initially, it may seem that integrating two applications together via a command pipeline would be difficult to implement and unwieldy to administer. Instead of developing a single application which shares all the necessary information internally, the designer must instead maintain two separate applications, and ensure that they are both able to understand each other's protocol when it comes time for the two software units to share information. Despite this apparent drawback, there are several noticeable benefits from this strategy.

From a purely software engineering perspective, splitting the implementation into two separate units enforces very loose coupling between the two applications, thereby making each of the units more autonomous and cohesive. The two software units do not even share a single global variable between them. As a result, significant changes can be made in one module without adversely affecting the other. By making the two units separate, we only share as much information as necessary between them via the command pipeline. Had the two modules been tightly bound together, the temptation to share numerous data structures between them would be very strong. Subsequent changes to these data structures would have repercussions that would permeate throughout the entire implementation. By modularizing the GUI and the simulator engine, the effects of radical implementation changes in one module do not usually extend into the other module, hence localizing the impact of the changes.

By creating a clear implementation distinction between the GUI and the simulator engine, it becomes much easier to remove one of the modules and replace it with another that has similar capabilities. For example a different GUI written in an entirely different language may be placed on top of the existing simulator engine. The two will still be able to communicate with one another as long as the new GUI provides the simulator engine with the necessary details regarding the structural design of the circuit and can parse the output waveform results generated by the simulator engine.

Alternatively, instead of replacing modules, we could augment the system relatively seamlessly by adding new modules. For example, if one were doing a comparison of two different discrete-event simulators, the same GUI could be employed to invoke either of the simulators. It is a trivial matter to enhance the GUI to invoke one of several possible simulator engines.[2] Of course, any new modules added to the environment would have to be made to conform with the four phase communication process described in Section 5.4.

Another advantage of rigidly partitioning the GUI and the simulator engine into two distinct applications is that it establishes a framework by which the two units may eventually communicate with each other at the socket level. This could lead to several benefits since this implies that the GUI and the simulator engine do not necessarily have to be running on the same machine. For example, several people could conceivably be working on the design of a different circuit subsystem using different GUIs on very limited graphics terminals. The simulator engine they all use, however, could be running on a high-powered machine which is accessible to all. The transmission of the structural circuit information to the so-called "simulation server" would be transparent to the end user. With socket support forthcoming in the Tcl core, this option of distributing the GUI and the simulator engine over several machines will become realizable. Of course, an efficient distribution of the GUI and simulator engine over several machines would have to take into consideration the potential communication delays inherent within any network. Similarly, effectively distributing the circuit representation over several machines for simulation would necessitate an intelligent partitioning of the circuit's subcomponents in such a way so as to minimize network traffic between machines.

[2]Note that if doing continuous simulation at the transistor level, the GUI will require changes to support the drawing of transistor symbols on the workarea canvas and the signal display would have to be adapted to support continuous waveforms.

5.3 Overview of the Interaction Protocols

The protocol for the previous example was very trivial — the Tcl script supplied two integers to the pipe, the C++ program read the two numbers and returned the sum back to the Tcl program via the pipe. This section will explain the more sophisticated protocols required by the GUI and the simulator engine to interact with one another. A specific example will be presented in the next section to help clarify the details regarding the protocols.

Because Tcl cannot yet handle binary data, the stream of information transferred between the GUI and the simulator is pure *ASCII* text. The information transferred through the pipe is broken down into individual *stanzas*. The structure of each stanza is composed of a *stanza header* and a *stanza body*. The stanza header can be thought of as representing a single entity, such as a component or a signal and consists of a single word followed by a colon. The stanza body contains attribute/value pairs which serve to qualify the entity. Such attributes include the type of the component or the list of values and times for a signal. Within each line of a stanza body, a colon must be placed after the attribute, thereby separating it from its value and each line of the stanza body must be indented by a tab character so that it can be distinguished from a stanza header line. There are no restrictions placed upon the order of the lines in the stanza body, but all attributes must be specified. An example of the generic format of each stanza is as follows:

$$\text{Stanza}\left\{\begin{array}{l} \left.\begin{array}{lll} \textit{header}: & & \end{array}\right\}\text{Stanza Header} \\ \left.\begin{array}{lll} & \textit{attribute}_1: & \textit{value}_1 \\ & \textit{attribute}_2: & \textit{value}_2 \\ & \ldots & \\ & \textit{attribute}_n: & \textit{value}_n \end{array}\right\}\text{Stanza Body} \end{array}\right.$$

This textual protocol format is very easy to parse using Tcl and C++. This format is also quite simple to extend — if an entity requires another attribute, then another line is simply added to the stanza body and the module parsing the stream is modified accordingly to parse the new attribute line. A textual data stream, as opposed to a binary stream, also aids tremendously in the debugging process.

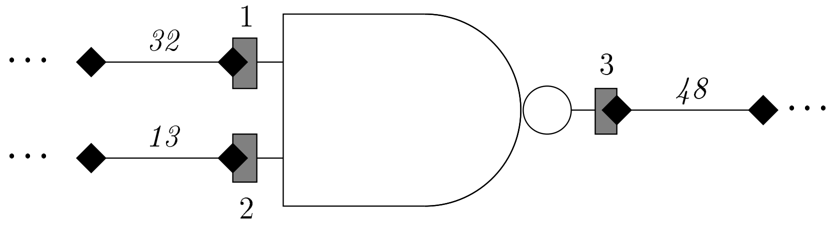
The interaction between the GUI and the simulator engine is comprised of two major protocols; namely, a *component protocol* which transfers the functional units from the GUI to the simulator engine and a *netlist protocol* which transfers the connectivity information from the GUI to the simulator. The netlist protocol is also responsible for transferring input and output signal values back and forth between the GUI and simulator engine. These two protocols form the basis of discussion for the next two subsections.

5.3.1 The Component Protocol

The header of a component stanza consists of the word `component`. Each component stanza has three attributes. The `type` attribute determines the type of the component; acceptable values include one of `and`, `nand`, `or`, `nor`, `xor`, `xnor`, `buffer` or `not`. The `id` attribute is used to distinguish among several components of the same type. Its value is a sequential serial number assigned to the component by the GUI. The `port` attribute is used to express the connectivity of the circuit in the context of its adjacent netlists. The value of this attribute is a list of integers, each of which represents a netlist connected to each port of the component. Therefore the number of integers in this list will be the same as the number of ports. The ports of a component are sequentially ordered by the GUI, with the input ports usually numbered before output ports. The position of the netlist numbers determines to which port the netlist is connected.

For example, consider the *NAND* gate in Figure 5.4. The two input ports are numbered 1 and 2, and the output port is numbered 3. This numbering is done internally by the GUI and is usually of no consequence to the end user. Netlist number 32 is connected to port 1, netlist 13 is connected to port 2 and netlist 48 is connected to port 3. If this *NAND* gate was assigned the serial number 7 by the GUI, then the stanza representing this component would be as shown below the *NAND* gate in Figure 5.4.

Figure 5.4: A 2-input *NAND* Gate and its Corresponding Protocol Stanza




```
component:
    type: nand
    id: 7
    ports: 32 13 48
```

5.3.2 The Netlist Protocol

The netlists comprising a circuit are represented by stanzas with the headers `input`, `output` or `internal`, each of which represent a different type of netlist. An `input` netlist is defined to be a netlist which is connected only to the input ports of one or more components; an `output` netlist is defined to be a labelled netlist which is connected to the output port of a component (an `output` netlist may also be connected to one or more input ports).[3] An `internal` netlist is the same as an `output` netlist, except that it has not been labelled by the user. Netlists which are not connected to the ports of any component are not transmitted by the protocol. The number of attributes each netlist has depends upon the netlist type and which module (either the GUI or the simulator engine) is generating the protocol. All three netlist types have an `id` attribute whose value is the netlist number as assigned by the GUI.

The GUI can generate stanzas representing all three netlists as part of its protocol to the simulator engine. Only the `id` attributes of the `internal` and `output` netlists are specified. The input netlists, however, each contain an additional attribute called `values`. The value of this attribute consists of a Tcl-like list of input signal values and the times that the signals occurred. The list takes the form of $\{t_0\ v_0\}\ \{t_1\ v_1\}\ \ldots \{t_n\ v_n\}$, where t represents the time of the signal and v represents its value — either *0*, *1* or *X*. In order to minimize the amount of data being transferred through the pipe, only the changes in the input signals are actually transmitted by the GUI to the simulator engine.

After constructing the circuit internally and initializing the input netlists according to the stanzas received from the GUI, the simulator engine then simulates the circuit, thereby producing signals on the output netlists. The output netlists are then traversed by the simulator engine. Stanzas representing each output netlist and their

[3]The input and output netlists are analogous as the primary inputs and primary outputs of the circuit.

corresponding identifiers and signals values are transmitted back to the GUI along the pipe for presentation. The format of the output stanzas produced by the simulator engine is identical to that of the input stanzas produced by the GUI. Again, only changes in output signal values are reported by the simulator engine so as to limit the volume of information sent along the pipe.

Examples of the stanzas comprising both the component and netlist protocols are presented in the next section which provides more implementation details regarding the communication between the GUI and the simulator engine.

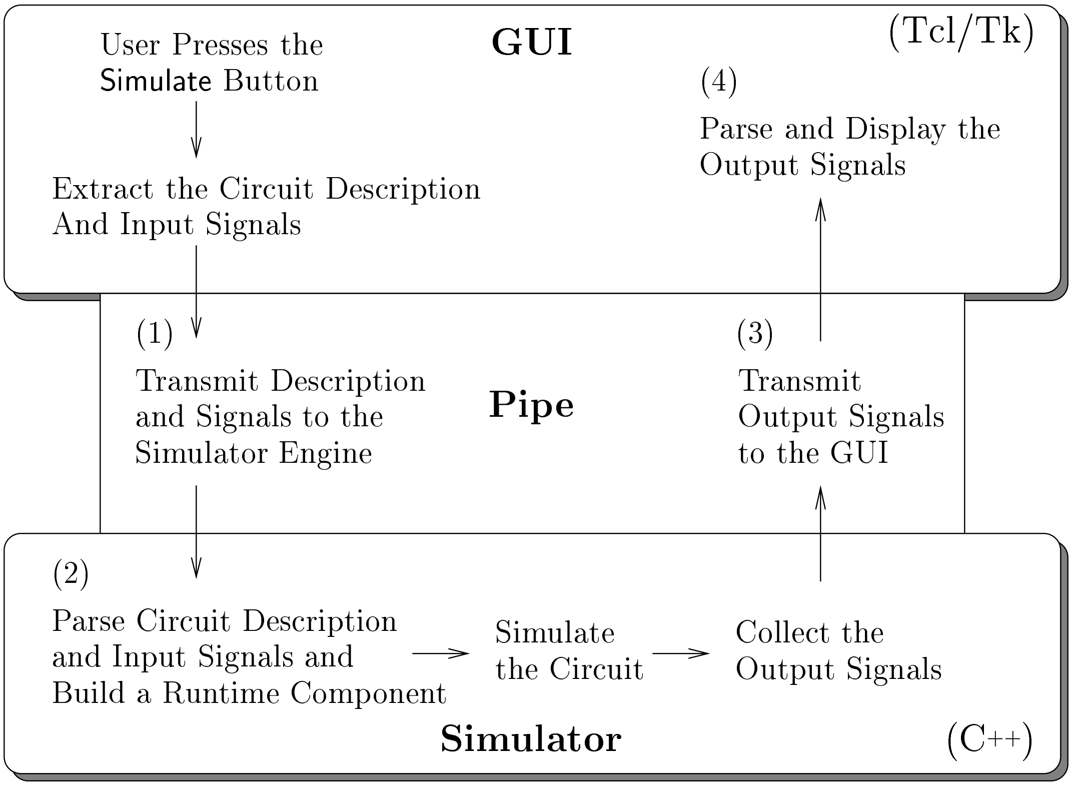
5.4 Implementation of the System Integration

This section discusses the details pertaining to the implementation of the interaction between the GUI and the simulator engine. As first introduced in Figure 5.1, the interaction takes place in four different steps; these steps are presented in greater context and in more detail in Figure 5.5. The GUI handles two of these steps and the simulator engine handles the other two. The subsequent sections will describe the implementation of these four stages in the context of a specific example.

5.4.1 Step 1: GUI Protocol Transmission

The first step involves the GUI transmitting component and netlist protocols to the simulator engine. These protocols describe the structure and connectivity of the circuit to be simulated and also provide the simulator with the input signals to be fed into the circuit during simulation. The Tcl procedure responsible for extracting the circuit description is called `simulate` which resides in the module `simulate.tcl`.

Figure 5.5: Intermodule Communication Between the GUI and Simulator



This procedure translates the component, netlist and input waveform information available on the workarea canvas and signal display window of the GUI into the component and netlist protocols described earlier. The stanzas comprising this protocol are then passed through the command pipeline opened by the `simulate` procedure and transmitted to the simulator engine.

For example, consider the circuit of Figure 5.6 and its corresponding input waveforms in Figure 5.7, both of which have been specified by an end user via the GUI. Note that the italic numbers in the circuit diagram represent the netlist numbers that identify the input, output and internal netlists connected to the component ports; they are presented as an aid to understanding the component protocol described below and do not appear to the end user on the GUI.

Figure 5.6: Example of a Circuit

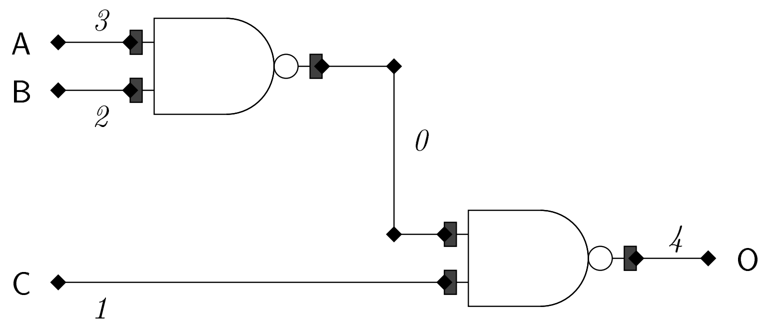
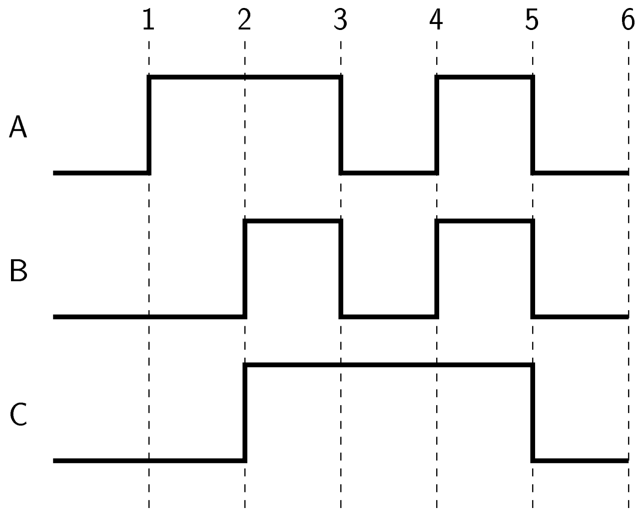


Figure 5.7: Example of Input Signal Waveforms



The stanzas representing the components, and the input, output and internal netlists are coalesced into the stream of data presented in Figure 5.8. Note that only the input netlists actually have corresponding signal values; the output and internal netlists are represented only by their respective identifiers. These stanzas are then transmitted to the simulator engine through the command pipeline.

Figure 5.8: Protocol for the Circuit and Inputs in Figure 5.6 and Figure 5.7

```
input:
    id: 3
    values: {0 0} {1 1} {3 0} {4 1} {5 0} {6 0}
input:
    id: 2
    values: {0 0} {2 1} {3 0} {4 1} {5 0} {6 0}
input:
    id: 1
    values: {0 0} {2 1} {5 0} {6 0}
output:
    id: 4
internal:
    id: 0
component:
    type: nand
    id: 0
    ports: 3 2 0
component:
    type: nand
    id: 1
    ports: 0 1 4
end
```

5.4.2 Step 2: Simulator Protocol Reception

During the second step, the simulator engine must parse the stanzas it receives from the GUI, construct the corresponding circuit and then simulate it. The two major C++ modules involved in this step are `parser.cpp` and `rtcomp.cpp`. The former module implements a class that provides support for reading the stanzas the simulator engine receives and then identifying the headers and the attribute/value lines in their bodies. This module makes heavy use of the second module, `rtcomp.cpp`, which stands for *runtime component*. This module implements a high-level component class which is built during runtime as the parser module interprets its input stream.

From a design perspective, a runtime component is *a kind of* component, so we derive the Runtime_Component class from the Component class, thereby enabling the Runtime_Component to inherit all the features of its base Component class. The Runtime_Component module extends the functionality of its base class by providing member functions which create the subcomponents and top-level ports which comprise the circuit to be simulated. In many cases, these member functions are simple wrappers around the port and component constructors themselves.

Continuing our example from the previous subsection, the stanzas shown in Figure 5.8 are received by the parser of the simulator engine. As the parser processes each of the stanza headers and their respective bodies individually, it invokes the appropriate methods of the Runtime_Component to create the necessary entities of the circuit. First, the primary input and primary output netlists of the circuit are constructed from the input and output stanzas transmitted by the GUI. Internal netlists are similarly constructed. For the input netlists, the input vectors of the `input` stanza body are also parsed and placed in the input netlist wire queues. Then, as the `component` stanzas are read, the subcomponents comprising the circuit are created. Note that the netlists are transferred to the simulator engine before the components because the constructors for components accept, as parameters, netlists.[4] Therefore, the netlists must be constructed before the subcomponents can be built.

After construction of all the netlists and subcomponents, the `simulate()` message is sent to the runtime component by the `main()` program, thereby simulating the newly constructed runtime component. All the resultant output signals are collected

[4]More accurately, the component constructors accept references to Connector objects as parameters, which are then connected to the ports of the components.

from the output netlist wire queues which are then transmitted by the simulator engine to the GUI for visual presentation in the signal display window. The `main()` function of the simulator engine is presented in its entirety in Figure 5.9. The mechanism by which these output signals are gathered and transmitted back to the GUI is described in the next subsection.

Figure 5.9: The `main()` Function of the Simulator Module

```
int main()
{
    Runtime_Component   component;
    Parser              parse;

    if (parse.ckt(component) != 0)
        return -1;
    component.simulate();
    component.show_outputs();
    return 0;
}
```

5.4.3 Step 3: Simulator Protocol Transmission

In order to transfer the output signals from the runtime component to the GUI, the simulator module must retrieve all the signal values on each of the output netlists of the runtime component. As seen from Figure 5.9, this is accomplished by the member function `show_outputs()`, which Runtime_Component inherits from its base class Component. This member function simply iterates over all the output ports of the runtime component and sends each of the output ports the message `show_signals()`, which they inherit from the Port base class.

As with the `get_signal()` and `send_signal()` member functions described earlier, the `show_signals()` messages eventually reach the output ports' respective wires, causing the `show_signals()` member function of each of the wires to be invoked. This Wire member function is presented in Figure 5.10. The function simply sends, to standard output, the `output` stanza header, the `id` attribute followed by the identifier of the netlist as well as the `values` attribute.

Figure 5.10: The `show_signals()` Member Function of the Wire Class

```
void Wire::show_signals() const
{
    cout << "output:" << endl;
    cout << "\tid: " << get_name() << endl;
    cout << "\tvalues: ";

    display_signals();
    cout << endl;
}
```

It then calls the member function `display_signals()` to actually retrieve and display the values of the signals and the times that they occurred during the simulation. This function is presented in Figure 5.11 and simply amounts to a traversal of all the signals on the wire. The output operator, `<<`, has been overloaded by the Signal class to output the signal time and value enclosed in braces. Note that only the changes in the output signals are sent to standard output by this method.

In the context of our particular example, because the circuit had only one output signal, only one `output` stanza is sent to standard output by the simulator engine. The stanza representing this output waveform is presented in Figure 5.12. Note that

Figure 5.11: The display_signals() Member Function of the Wire Class

```
void Wire::display_signals() const
{
    List_Iterator<Signal>    sigs(signals);
    Signal                  *sig;
    Sig_Val                  prev_val = SIG_X;
    int                      num = signals.num_elements();
    boolean                  first = TRUE;

    while ((sig = sigs()) != 0)
    {
        num --;
        if (first || sig->get_value() != prev_val || num == 0)
        {
            cout << *sig;
            prev_val = sig->get_value();
        }
        first = FALSE;
    }
}
```

the time and value of the first signal is {_ X}. This represents the initial time and initial value of the output signal. Because of the command pipeline established by the GUI, all the output signals that the simulator engine generates on standard output will be picked up by the GUI, parsed and then displayed graphically. This final step is described in the next subsection.

Figure 5.12: Sample Protocol for an Output Signal Waveform

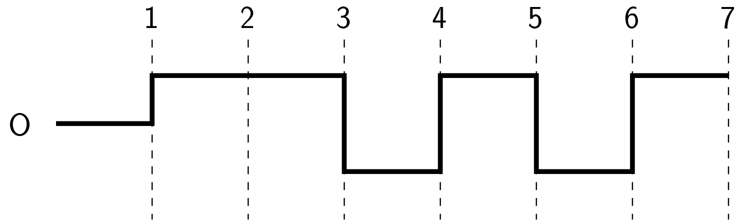
```
output:
    id: 4
    values: {_ X} {1 1} {3 0} {4 1} {5 0} {6 1} {7 1}
```

5.4.4 Step 4: GUI Protocol Reception

Once the simulator engine has finished executing and has transmitted the output signals back to the GUI, the `simulate.tcl` module of the GUI takes control again. The `sim_process_outputs` procedure of this module is invoked to parse all the signal times and values it received from the simulator engine. For each output netlist, `sim_process_outputs` calls the `sig_draw_signal` procedure of the `sigdisp.tcl` module to display the corresponding output waveform in the signal display window. Upon completion, the file identifier for the command pipeline is closed and the GUI will once again respond to input from the end user.

In the example from the previous section, the waveform corresponding to the protocol in Figure 5.12 is shown in Figure 5.13. After examining the waveform, the user may change the input signals or the circuit itself and then re-simulate the circuit until the desired output waveform is achieved.

Figure 5.13: Output Signal Waveform Displayed Graphically



In summary, the GUI may be thought of as the structural subsystem of the entire architecture, in that it provides the means by which components are interconnected and structurally defined by the user. The simulator engine itself may be thought of as the functional or behavioural subsystem, since it is responsible for mirroring the circuit description provided by the user and injecting each of the components with behavioural and functional semantics necessary to give life to the circuit.

Chapter 6

Conclusions

This report has discussed the design and implementation of a complete environment for designing and simulating simple digital circuits at the logic level. By adopting sound design principles and by employing a high-level user interface toolkit, an intuitive front-end GUI was implemented for the layout component of the application. The creation of the GUI has dramatically increased the accessibility of the underlying simulator engine by enabling the user to interact with the circuit graphically instead of textually. The GUI also enables relatively complicated circuits to be created and simulated, thereby providing a more efficient means for verifying the correctness of new simulator engines.

The simulator engine described in this report employs distributed event queues for the purposes of event notification and signal propagation instead of the more conventional single global event queue. By using distributed queues, the circuit model more accurately represents its counterpart in the real world. The implementation of the simulator strategy was facilitated by the object-oriented paradigm. The support for encapsulation, inheritance and polymorphism provided by this paradigm made the implementation more natural and more resilient to change. The component class library was designed to be extensible; consequently, a viable foundation has been established upon which additional circuit elements can be derived and incorporated into the simulator engine, resulting in a more comprehensive component class library.

By cleanly separating the GUI from the simulator engine and having them communicate via a bidirectional pipe, future extensibility and reuse of the two software modules is realizable. In addition, a new GUI or new digital simulator engines can be added relatively seamlessly to the environment, provided they each conform to a mutual protocol for the purpose of information sharing. The clear distinction between the GUI and the simulator also makes it easier to modify the implementation of one of the modules without adversely affecting the other. The partitioning of the two modules may also serve to help distribute the execution of the software over several machines.

6.1 Applications and Future Work

With respect to potential uses of the software, the current implementation has academic merit and may be useful in an introductory course on digital logic. From a research perspective, the GUI can be used to give a common interface for other simulator engines, thereby providing a uniform environment in which the performance of several different digital simulator engines may be compared. However, despite the potential benefits and applications of the GUI and simulator, there are still several improvements which can be made to enhance the functionality and practicality of both modules. Some of these enhancements could lead to industrial applications of the software.

In the context of the GUI, perhaps the biggest drawback is the lack of hierarchical support in the circuit layout editor. While the simulator engine itself does support arbitrary nesting of components, the GUI, unfortunately, still forces the user to adopt a flat view of the circuit being designed. Once hierarchical support is implemented for the GUI, the existing protocol between the GUI and the simulator would have to be enhanced to support the multi-level nature of the circuit design. Another limitation is the lack of component configurability from the GUI. For example, the GUI does not yet have the ability to let the user specify the number of inputs for a logic gate. For the most part, the user is restricted to using two input gates. Also, the end user should be able to specify the component delay using the GUI. One final minor enhancement is to have the GUI report all of its diagnostics and error messages to a separate modal dialog box. Currently, most diagnostics are sent to standard output, which is usually hidden by the circuit editor window. By presenting the diagnostics in a separate window, or maybe even in a subframe of the main window, the warning and error messages generated by the GUI will become more obvious to the user.

There are still some internal issues left to resolve with respect to the existing GUI code base. For example, as the implementation progressed, the prefixing convention for procedure and variable names became more difficult to maintain. Also, the required proliferation of global variables and arrays in the source code compromised the level of encapsulation between Tcl modules. In the future, a possible rewrite of the GUI using an Tcl/Tk extension language with better namespace control and more effective code sharing support may be possible. However, the disadvantage with using extension packages is that they may not always keep pace or be compatible with the latest release of the Tcl/Tk core from Sun Microsystems.

With upcoming releases of the Tcl core supporting nonblocking I/O, it may be possible to make the simulation interactive. This would permit the user to pause and maybe even reverse the simulation while it is in progress, for example. It may also be possible to animate the simulation itself, thereby demonstrating to the user the propagation of signals along netlists, hence making the circuit behaviour more lucid.

With the rise in the popularity of the Internet and the World Wide Web, one future project might be to rewrite the simulator engine using the Java programming language and to then integrate it with the Tcl/Tk GUI. This would enable anyone with a Java compliant web browser that has the appropriate Tcl/Tk bindings to download and execute the GUI and simulator directly over the Internet. How this would actually be accomplished depends largely upon how the two languages can be integrated with one another.

From the point of view of the simulator engine, there are many enhancements which would increase the potential usefulness of the engine. For example, zero delay components are not yet feasible with the current state of the implementation. In theory, however, it does seem possible to implement zero delay elements in a distributed queue simulator. If a component sends a signal to one of its output wires that conflicts with an existing signal on the wire at the same time, the component would not update its local time. If the output signal does not stabilize, then after a given number of oscillations, the simulation will be terminated with a diagnostic, indicating a possible flaw in the circuit design.

Like the GUI, the circuit representation employed by the simulator engine restricts the number of inputs of the basic logic gates to two. A more generic mechanism for specifying the number of inputs to these gates would contribute significantly to the extensibility of the engine. This can be achieved by supplying the appropriate component constructors with an array of connector references instead of passing the connector references individually. The addition of a clock class would also improve the simulator engine, since it would facilitate the creation of synchronous circuits. When queried by a component which has a clock input, the clock object would return the signal value that would occur at the time requested by the component in accordance with the frequency of the clock.

These enhancements would serve to increase both the academic and industrial viability of this software package and may even serve as the foundation upon which more generic queuing simulators may be based.

Bibliography

[1] Augustin, Larry M., David C. Luckham, Benoit A. Gennart, Youm Huh and Alec G. Stanculescu, *Hardware Design and Simulation in VAL/VHDL*, Kluwer Academic Publishers, Boston, Massachusetts, 1991.

[2] Billard, Edward A. and Alice E. Riedmiller, "Q-Sim: A GUI for a Queueing Simulator Using Tcl/Tk," *ACM SIGSOFT*, pp 82–84, Vol. 19, No. 4, October 1994.

[3] Booch, Grady, *Object-Oriented Analysis and Design with Applications, 2nd Edition*, Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[4] Cellier, François E., *Continuous System Modeling*, Springer-Verlag, New York, New York, 1991.

[5] Chandy, K.M. and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, pp 198–206, Vol. 24 No. 11, April 1981.

[6] Coelho, David R., *The VHDL Handbook*, Kluwer Academic Publishers, Boston, Massachusetts, 1989.

[7] Craig, Donald C., "Circuit Description and Elementary Hierarchical Circuit Simulation Using C++ and the Object Oriented Paradigm", Bachelor of Science Dissertation, Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada, A1B 3X5, 1991.

[8] Cloutier, J., M. Bourgault, S. Fauvel, C. Roy, E. Cerney and J. Gecsei, "An Object-Oriented Mixed-Mode Hierarchical VLSI Simulator," Dept. d'information et de recherche operationnelle, Université de Montreal, 1986.

[9] d'Abreu, Manuel A., "Gate-Level Simulation," *IEEE Design & Test*, pp 63–71, Vol. 2, No. 6, December 1985.

[10] Gajski, Daniel D., editor, *Silicon Compilation*, Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1988.

[11] Gajski, Daniel D., Nikil D.Dutt, Allen C-H Wu and Steve Y-L Lin, *High-level Synthesis : Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, Massachusetts, 1992.

[12] Hogan, Sean M., "YADIS–1 — An Implementation," Technical Report #8906, Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada, A1B 3X5, 1991.

[13] Kocher, Hartmut and Martin Lang, "An Object-Oriented Library for Simulation of Complex Hierarchical Systems," *Object Oriented Simulation Conference (OOS'94): Proceedings of the 1994 Western Multiconference*, Society for Computer Simulation, San Diego, California, pp 145–152, Vol. 26, No. 2, 1994.

[14] Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language, 2^{nd} Edition*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[15] Linton, Mark A., John M. Vlissides and Paul R. Calder, "Composing User Interfaces with InterViews," *IEEE Computer*, pp 8–22, Vol. 22, No. 2, February 1989.

[16] Mak, Victor W., "*DOSE*: A Modular and Reusable Object-Oriented Simulation Environment," *Object Oriented Simulation 1991: Proceedings of the SCS Multiconference on Object-Oriented Simulation*, Society for Computer Simulation, San Diego, California, pp 3–11, Vol. 23, No. 3, 1991.

[17] Mead, Carver and Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1980.

[18] Ousterhout, John K., *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1994.

[19] Peddie, Jon, *Graphical User Interfaces and Graphic Standards*, McGraw Hill, Inc., New York, New York, 1992.

[20] Popken, Douglas A., "An Object-Oriented Simulation Environment for Airbase Logistics", *Simulation*, pp 328–338, November 1992.

[21] Quercia Valerie and Tim O'Reilly, *X Window System User's Guide*, O'Reilly & Associates, Inc., Sebastopol, California, 1993.

[22] Randhawa, Sabah U., Charles C. Brunner, James W. Funck and Guangchao Zhang, "A Discrete-Event Object-Oriented Modeling Environment for Sawmill Simulation," *Simulation*, pp 119–130, February 1994.

[23] Ripley, Brian D., *Stochastic Simulation*, John Wiley & Sons, New York, New York, 1987.

[24] Ruehli, A. E., editor, *Circuit Analysis, Simulation and Design*, Elsevier Science Publishing Company, Inc., Amsterdam, North-Holland, 1986.

[25] Rubin, Steven M., *Computer Aids for VLSI Design*, Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1987.

[26] Shannon, Robert E., *System Simulation: The Art and Science*, Prentice-Hall, Englewood Cliffs, New Jersey, 1975.

[27] Sommerville, Ian, *Software Engineering, 4th Edition*, Addison-Wesley, Montreal, Quebec, 1992.

[28] Stroustrup, Bjarne, *The C++ Programming Language, 2nd Edition*, Addison-Wesley Publishing Company Inc., Reading, Massachusetts, 1991.

[29] Welsh, Matt and Lar Kaufman, *Running Linux*, O'Reilly & Associates, Inc., Sebastopol, California, 1995.

[30] Wolf, Wayne H., "Object-Oriented Programming for CAD," *IEEE Design & Test*, pp 35–42, Vol. 6 No. 5, March 1991.

[31] Zeigler, Bernard P., *Multifacetted Modelling and Discrete Event Simulation* London, Academic Press, 1984.

[32] Zeigler, Bernard P., *Object-Oriented Simulation with Hierarchical, Modular Models : Intelligent Agents and Endomorphic Systems* Boston, Massachusetts, Academic Press, 1990.

Appendix A

Installation Guide

This appendix provides information related to the installation and use of the circuit editor GUI (hereafter referred to as *DigiTcl*) and the simulator engine, as described in this report. These instructions are valid at the time of writing, but may not be valid for subsequent releases of the software. At the time of writing, the current version of *DigiTcl* is 0.1.0 and should be considered a *beta* release.

These installation instructions assume that the target machine is running a recent release of the *X* Window System (Version 11), on a *UNIX*-like operating system. Tcl 7.4 and Tk 4.0 (preferably patch level 3) must also have been installed on the target machine before *DigiTcl* can be used.[1] A C++ compiler is also required so that the simulator engine can be compiled. Version 1.2.4 of the GNU `gzip` compression utility must also be available on the machine so that the archive file can be decompressed before being extracted. The standard complement of *UNIX* utilities will also be required. The package may be installed and used without root privileges,

[1]Although later versions of Tcl/Tk have been ported to other non-*UNIX* operating systems, *DigiTcl* has not yet been tested on any of them.

if necessary.

A.1 Extracting the Archive File

The distribution includes a single compressed archive file called `digitcl-010.tar.gz` which can be extracted using the command:

```
$ gzip -dc digitcl-010.tar.gz | tar xvf -
```

This will create a directory called `digitcl-010` and all the relevant files will be placed in that directory. These files and subdirectories are described in Table A.1.

Table A.1: Files Included in the *DigiTcl* Distribution

Files	Description
`README`	A text file with instructions on how to install and use *DigiTcl*.
`digitcl`	The Tcl/Tk script for the *DigiTcl* GUI circuit editor.
`bitmaps/*`	Bitmaps and mouse cursors required by the *DigiTcl*.
`cktsim/*`	The C++ source and `Makefile` for building the simulator engine.

A.2 Compiling the Simulator Engine

In order to simulate circuits, the simulator engine must be compiled. The GNU C++ compiler, version 2.7.2, should be able to successfully compile the engine. Unfortunately, there are known template instantiation problems associated with version 2.6.3 of the same compiler.

To compile the simulator engine, simply change to the `cktsim` directory and type `make`. If there were no errors during the compiling and linking, a `digisim` executable should be created in the directory. This binary will be executed as a subprocess by *DigiTcl* — there is no need to run the executable directly.

A.3 Environment Variables

Before *DigiTcl* can be used, the appropriate shell variables must be set and exported to the environment. Some of these environment variables may be required by the underlying graphics toolkit, Tcl/Tk (see Table A.2), while others are required by *DigiTcl*, itself (see Table A.3). The standard `PATH` environment variable may also be modified for user convenience.

Table A.2: Environment Variables Used by Tcl/Tk

Environment Variable	Description
`TCL_LIBRARY`	Set to the location of the Tcl library files. It is not necessary to set this environment variable if `tclsh` was compiled with the library path pre-configured in its binary.
`TK_LIBRARY`	Set to the location of the Tk library files. Again, it is not necessary to set this environment variable if `wish` was compiled with the library path pre-configured in its binary.

Table A.3: Environment Variables Used by *DigiTcl*

Environment Variable	Description
`DIGISIM`	Set to the location of the simulator engine compiled earlier. An absolute pathname should be used.
`DIGILIB`	Set to the location of auxiliary bitmap and cursor files required by the circuit editor. Again, an absolute pathname should be used.
`DIGIUSER`	Set to the location of the working directory to be used by `digitcl`. When loading or saving circuits, the contents of the directory indicated by this environment variable will be presented in the file selection dialog box.
`PATH`	This standard environment variable should include the location of the `wish` binary. It may also contain the location of the `digitcl` script.

A.4 Running the *DigiTcl* Circuit Editor

Once the simulator engine has been compiled and the environment variables have been set, the *DigiTcl* circuit editor may be started by typing `digitcl` while in the `digitcl-010` directory. For convenience, the full path of the `digitcl-010` directory may be added to the `PATH` environment variable, thereby letting the user execute the script while in any directory.

If the location of the `wish` executable was not included in the user's `PATH` or if the installed `wish` binary is not actually named `wish`, then the circuit editor must be invoked by specifying the full path of the `wish` binary and providing the pathname of the `digitcl` script as an argument. For example:

```
$ /usr/local/bin/wish4.0 ~/digitcl-010/digitcl
```

If the `DIGIUSER` environment variable was not set, then the contents of the directory that was current when `digitcl` was invoked will be presented in the file selection dialog box. The file selection dialog box is displayed whenever the user activates the Open..., Save... or Save As... options of the File pull-down menu.

Unfortunately, a `Makefile` that installs the distribution files in standard directories does not yet exist. Any changes to the locations of the files will have to be made in accordance with the installation information provided above. In particular, the environment variables have to be modified appropriately when moving the location of the *DigiTcl* script, the simulator engine executable or the auxiliary files required by *DigiTcl*.

Appendix B

Circuit File Format

This appendix describes the file format used to store circuits created by the circuit editor GUI, thereby achieving persistence across several editing sessions. When the user saves a circuit to a file using the Save... or Save As... options of the File pull-down menu, all the details relating to the location of the wire points and the placement and orientation of the components are stored in the file. The netlist labels as well as the signals values on each of the labelled netlists are also saved in the file. After a circuit is saved, it may be restored during subsequent editing sessions using the Open... option.

In order to save circuits to a file, a circuit description language which accurately reflects the detailed structure of the circuit was devised. This description language is similar in style to the component and netlist protocols described in Chapter 5, since they both adopt a stanza format for the circuit description. Unlike the component and netlist protocols, however, the circuit file format is much lower level and contains detailed structural information relating to the placement of components and wire points.

The language which describes the physical layout of the circuit is composed of three types of stanzas; namely `component`, `point` and `label` stanzas. Each of these stanzas are described in subsequent sections.

B.1 The `component` Stanza

There is one `component` stanza for each component in a circuit. Stanzas representing components have four attributes; namely, `type`, `coords`, `orient` and `ports`.

The `type` attribute, as in the component protocol transmitted to the simulator engine, represents the type of the component. Current acceptable values for this attribute include `and`, `nand`, `or`, `nor`, `xor`, `xnor`, `buffer` or `not`. The `coords` attribute stores the coordinates of the center of the component on the canvas as a pair of integer coordinates. These coordinates are obtained from the `cmp_coord` global array in the GUI implementation. The `orient` parameter stores the orientation of the component and may be set to one of `0`, `90`, `180`, or `270`, each of which represents the number of degrees of rotation. Finally, the `ports` attribute specifies to which wire points the ports of the component are connected. The value of this attribute is a Tcl-like list which consists of pairs of elements. The first element is a concatenation of the port type, either input (`i`) or output (`o`), and the sequence number of the port. The second element of each pair is the identifier of the wire point to which the port is attached. An example of a `component` stanza is presented in Figure B.1.

Figure B.1: Example of a `component` Stanza

```
component:
    type: nor
    coords: 320 190
    orient: 0
    ports: {i000 211} {i001 213} {o000 205}
```

B.2 The `point` Stanza

As with the `component` stanzas, there is exactly one `point` stanza for each wire point on the workarea canvas. Each `point` stanza has four attributes; namely, `id`, `coords`, `adjcnt` and `net`.

The `id` attribute represents the unique numeric identifier of the point on the workarea canvas. It serves as a means by which the point can be referenced by other stanzas in the circuit description. The `coords` attribute, like the `coords` attribute of the `component` stanza, stores the coordinates of the wire point on the canvas. These coordinates are obtained from the `pnt_coord` global array of the GUI implementation. The `adjcnt` attribute stores the identifiers of all the points that are adjacent to the point. The final attribute, `net`, stores the identifier of the netlist to which the point belonged when the circuit was saved. This value is stored in the circuit file so that netlist extraction does not have to take place when the circuit is restored by the user. Note that there is no need to store the identifiers of wires in the circuit file since each of the wires can be reconstructed from the adjacency lists of each point. An example of a `point` stanza is presented in Figure B.2.

Figure B.2: Example of a `point` Stanza

```
point:
    id: 205
    coords: 360 190
    adjcnt: 222 254 201
    net: 5
```

B.3 The `label` Stanza

Each `label` stanza represents a netlist that has been labelled by the user. Again, there are four attributes associated with each `label` stanza. They are `name`, `point`, `anchor` and `values`. Each of these attributes are discussed below.

The `name` attribute simply specifies the name that the user assigned to the netlist in the Netlist Label Dialog box. It is composed of a string of characters. The `point` attribute indicates to which point the label is attached. Remember that to label a netlist, the user must label exactly one wire point that is a member of that netlist; the `point` attribute stores the identifier of the point that was selected by the user to label the netlist. The anchor attribute specifies the location of the label name relative to the point. It may be set to one of `n`, `ne`, `e`, `se`, `s`, `sw`, `w`, `nw` or `c`; where `n` is north, `s` is south, `w` is west, `e` is east and `c` is center. The `values` attribute stores the list of signals that were associated with the netlist. Note that if the netlist is an input netlist, then this list of signals was specified directly by the user. If the netlist is used for output, then the signal list was generated as the result of a simulation. As with the netlist protocol described in Chapter 5, the signal list takes the form $\{t_0\ v_0\}\ \{t_1\ v_1\}\ \ldots \{t_n\ v_n\}$, where t represents the time of the signal and v represents its value — either *0*, *1* or *X*. Only the changes in the signal values are actually stored.

An example of a `label` stanza is presented in Figure B.3.

Figure B.3: Example of a `label` Stanza

```
label:
    name: Q'
    point: 208
    anchor: w
    values: {0 X} {2 1} {5 0} {10 1}
```

From all of these stanzas, it is possible for the GUI to reconstruct the entire physical representation of the circuit on the workarea canvas when the user restores the circuit from the file using the Open... option of the File pull-down menu.

Appendix C

Simulator Engine Class Dictionary

This appendix provides an overview of all the C++ classes used by the simulator as described in Chapter 4. Each of these classes employ a well defined functional interface which together provide a small but powerful foundation that forms the basis of a distributed, hierarchical circuit simulator. These classes have been shown to be successful in the description and simulation of the following types of circuits:

- simple, atomic components, such as a 2-input *AND* gate,
- components comprised of other nested components, for example, a three-input *AND* gate constructed using two 2-input *AND* gates and a wire and
- components which contain feedback loops, such as an RS-Latch.

By using object-oriented techniques such as encapsulation and inheritance, a flexible, intuitive approach to digital logic hardware simulation is possible; objects instantiated from classes in the library bear close resemblance to their corresponding real world counterparts. The class library also provides support for a runtime component and an elementary parser which together can be used to interface the simulator engine with a graphical user interface.

The following sections contain a description of each class and the corresponding public, protected and private data members. In addition to the member functions described below, most of the classes also contain a public method to display diagnostic information regarding the current state of an object instantiated from the class. For brevity, destructors for the classes described below are not discussed.

C.1 The **Component** Class

The Component class acts as an abstract base class upon which all specific hardware components must be derived. The objects instantiated from these derived classes contain all the elements and functions to process input signals and produce appropriate output signals. If the component contains subcomponents, its purpose is to pass signals it receives to its encapsulated subcomponents.

C.1.1 Public Members

- `List<Port *> I_List, O_List`: These two members maintain linked lists of input and output ports respectively as required by the component. Each element of the linked list is a pointer to a Port object, which is described in detail in a subsequent section. These linked list of ports are used extensively during the simulation in order to coordinate the processing of inputs and outputs and to access lower levels of the component hierarchy built during the construction of the circuit. Because a component communicates with the external world via its ports, the port lists are made part of the public interface of the Component class. Note that this is one of the few instances in the implementation where public access is granted to the data members of a class.

- `void process(ckt_time time)`: This is a virtual method which implements the functional behaviour of the component. For high level components, (that is, components which are composed of subcomponents) this function scans over all the port pointers in the aforementioned input port list and activates all the subcomponents which are connected to these input ports. Components at the lowest level of the hardware hierarchy, however, must override this virtual function to provide the specific functionality of the component. For example, a 2-input *NAND* gate would examine its two inputs signals at the specified time and produce a low output only if both inputs are high.

- `void simulate()`: This function is employed by all classes derived from the Component class, regardless of the hierarchical level of the component. This method first determines if all the inputs for the component are ready at the local time of the component. If so, then this method will increment the local time and invoke the `process()` member function to trigger the component to consume its inputs and produce appropriate outputs. The `simulate()` method is called recursively as control travels down the three dimensional component object tree.

- `void show_outputs()`: In order to determine the results of the simulation, this function must be called. This method simply traverses the ports in the output port list of the component and displays all the signal values and times that are stored in the output wires connected to each of the ports.

C.1.2 Protected Members

- `Component(ckt_time delay, const char *name)`: This is the constructor for the component class. It accepts a timing value which represents the transport delay of the component and a string representing the name of the component. The constructor simply initializes the delay time, name and local time of the component. This member is made protected so as to prevent objects of type Component from being instantiated. Only objects derived from Component which have public constructors may actually be created.

- `ckt_time get_delay()`: This is a simple accessor function which returns the transport delay of the component. It is commonly used by the `process()` function of low-level components to determine how much time to add to output signals when generating outputs.

C.1.3 Private Members

- `boolean inputs_are_ready(ckt_time time)`: This is a helper member function that is used exclusively by the `simulate()` member function. When a component receives the `simulate()` message it must first determine if all its inputs are ready for its local time. The `inputs_are_ready()` method serves this purpose by sending the `get_signal()` method to all input ports in the input port list. If all the inputs are available at the specified time, this method returns `TRUE` and the `simulate()` method can continue; otherwise it returns `FALSE`.

- `ckt_time delay`: The simulator engine uses the transport delay model when simulating circuits. The `delay` data member is used to represent this delay. Note that, like the `process()` function, it is only necessary for components at the lowest level of the hierarchy to define a delay time since the delay of higher level components is dependent upon the delay time of its subcomponents.

- `char *name`: This member is used to store the name of the component. This is useful for identifying the component when debugging the simulator engine.

- `ckt_time local_time`: Due to the distributed queue approach adopted by the simulator, it is possible for several components to be at different times during the simulation. This data member represents the location of the component along the temporal range of the simulation. Only the `simulate()` method needs to access this member, hence it is made private.

C.2 The Connector Class

The Connector class is an abstract base class responsible for connecting components with other components and for connecting components with the external world. This class maintains a linked list of components which is representative of the fan-out list of the connector. This class is also used as a basis for signal communication during the simulation process. That is, whenever a component wishes to get or send a signal, it must do so through a connector. This class serves as a foundation upon which two other classes, namely Wire and Port are derived. The Port class, in turn, serves as an abstract base class for the Input and Output classes. These classes are discussed in detail in subsequent sections.

C.2.1 Public Members

- `Signal get_signal(ckt_time time)`: This is a pure virtual function which is responsible for retrieving the signal which occurred at the time specified by the parameter. The Wire and Port classes actually define the behaviour of this member function. Consequently, the description of its implementation will be presented in later sections.

- `void send_signal(Signal sig)`: Like `get_signal()` above, this too is a pure virtual function. Its major purpose is to propagate an output signal from a component to the components in its fan-out list. The details of this function will be discussed in further detail in the sections pertaining to the Wire and Port classes.

- `show_signals()`: Again, this is a pure virtual function. Upon invocation, this function will display all the signals that have passed through the connector during the course of the simulation. It is defined by the Wire and Port classes.

- `void connect(Component &cmp)`: This method adds the specified component to the list of components in the fan-out list of the connector. It makes use of the `add` member function of the generic List class.

- `const char *get_name()`: Each connector has a name which serves to identify that connector. This function simply returns a pointer to that name. It is used primarily by the Runtime_Component class when constructing a circuit from its protocol description.

- `void propagate()`: This method is invoked during simulation. Its purpose is to iterate over the components in the fan-out list and to send each of them a `simulate()` message, thereby forcing the components to consume their inputs (when possible) and to produce outputs. These outputs are subsequently propagated in the same manner.

C.2.2 Protected Members

- `Connector(const char *name)`: The constructor for the Connector class simply accepts a string which is used to initialize the name of the connector. This name can be retrieved later using the `get_name()` accessor function.

C.2.3 Private Members

- `List<Component *> fan_out`: As described above, each connector is responsible for propagating the signals it receives to the components to which it is connected. The `fan_out` member is simply a linked list of pointers to components in the fan-out of the connector. This list is traversed whenever the connector receives a `propagate()` message.

- `char *name`: Like the Component class, this data member simply stores the name of the connector for debugging purposes. It is also used to identify connectors during the construction of runtime components.

C.3 The Wire Class

The Wire class is used primarily to connect two components together. During the course of a simulation, the Wire class also caches all the signals that have passed through it. Queries by an Input or Output port for a signal must eventually be answered by a Wire object. Similarly, all signals sent by an Output port must eventually be received by a Wire object. The Wire class is derived from the Connector class.

C.3.1 Public Members

• `Wire(const char *name)`: The Wire constructor accepts a name which represents the wire and passes it to its base class, Connector for initialization. An initial signal is then added to the wire's signal list by the constructor using the `add_signal()` member function.

• `Signal get_signal(ckt_time time)`: During simulation, components must be able to obtain the value of a signal that travelled along the wire at a specified time. The `get_signal()` method performs this task by traversing its linked list of signals for the correct signal. Checks are made during the search to ensure that the client code is not attempting to look for a signal which has not yet occurred.

• `void send_signal(Signal sig)`: When the Wire object receives this message, it adds the signal to its signal list by calling its `add_signal()` member function. It then attempts to propagate the signal to all the components in its fan-out list by invoking the `propagate()` method of its base Connector class.

• `void show_signals()`: This method will display an `output` stanza header and the signal values and times currently residing on the wire by calling the private `display_signals()` function. It is invoked indirectly by the `show_outputs()` member function of the Component class after a simulation has been completed.

• `void add_signal(Signal sig)`: This method will append the specified signal to the signal list maintained by the wire. The implementation of this function ensures that the signal is in time-order with respect to the last signal in the signal list; if not, then a diagnostic warning will be displayed. This function is used during the initialization of the wire and by the `send_signal()` member function. It is also used by the Runtime_Component class to add the initial signals to the primary input wires during the construction of the component.

C.3.2 Protected Members

The Wire class has no protected members.

C.3.3 Private Members

- `List<Signal> signals`: This private member acts as a repository for all the signals which have been transmitted through the wire during the course of the simulation. It is simply a linked list of Signal objects.

- `void display_signals()`: This helper method employed by `show_signals()` iterates over the signal list and displays all the signal values and times currently on the wire. Only the changes in signal values are actually displayed by this function.

- `void replace(Signal sig)`: This function is used to replace a signal in the list with the specified signal. It is intended for use during the simulation of zero-delay elements but is not used by the current implementation.

C.4 The Port Class

The Port class is yet another abstract base class which forms the foundation for both the Input and Output classes. The Port class provides the means by which components communicate with the external world. After construction, each input port will know its external feeding connector and each output port will know which external connector to feed. As mentioned earlier, a linked list of input and output ports is maintained in the public interface of each component. These ports control signal flow to and from the component. The Port class is derived from the Connector class.

C.4.1 Public Members

• `Signal get_signal(ckt_time time)`: Quite simply, this method retrieves the signal which occurred at the specified time from the connector which feeds it by sending a `get_signal()` message to the port's encapsulated external connector. The `get_signal()` method will recurse through higher-level ports until a wire is reached; at which point, the signal list of the wire will be traversed and the desired signal, if found, will be returned.

• `Signal send_signal(Signal sig)`: This method is used to transmit the specified signal to the external connector of the port by sending the `send_signal()` message to the external connector. As with the `get_signal()` method, `send_signal()` will recurse through higher-level ports until the destination wire is reached. The signal will then be added to that wire. This method also propagates the signal to all the components in the fan-out of the port by invoking the `propagate()` member function which it inherited from its base Connector class.

• `void show_signals()`: This method sends the `show_signals()` message to the external connector of the port in order to display all the signals that have passed through the port during the course of the simulation. As with the previous two methods, recursion will occur until a wire connector is reached, at which point the signal list of the wire will be traversed and displayed. This method is called by the show_outputs() method of the Component class.

C.4.2 Protected Members

• Port(Connector &con, const char *name): This constructor assigns the connector reference to the external protected data member as described next and passes the name of the port to the base Connector constructor. The constructor of the Port class is protected so as to prevent actual objects of the class from being instantiated; only classes derived from Input and Output can be created.

• Connector *external: This member represents the external connector to which the port itself is connected. This data member usually points to a port in an hierarchical level immediately above the port or to a wire in the same hierarchical level as the port. Note that because Wire and Port are both derived from the Connector class, external may point to either a Wire object or a Port object.

C.4.3 Private Members

The Port class has no private members.

C.5 The **Input** Class

The Input class is derived from the Port class and is used to simulate an input port of a component. The input ports of a component are all stored in the I_List public member of the Component class and can therefore be accessed by outside objects as is the case in the real world. The primary purpose of an Input object is to permit the enclosing component to receive signals from the external feeding connectors in the hierarchy immediately above the port. The Input object is also responsible for notifying all the subcomponents which it feeds to process their inputs.

C.5.1 Public Members

- `Input(Component &cmp, Connector &con, char *name)`: In order to build an input port for the runtime component, the Input constructor takes a reference to the component which encloses the input port, a reference to the connector which feeds the input port and a name for the input port. The connector reference and name parameters are passed to the Port base class constructor, thereby connecting the port to its external source connector and storing the name of the port. A `connect()` message is then sent to the external connector with the component reference as a parameter. This adds the component which encloses the port to the fan-out list of the external connector. The input port is then added to the input port list of the enclosing component.

- `void send_signal(Signal sig)`: This method overrides the corresponding virtual function in the Port class. Because input ports cannot send signals, this function simply displays an error message indicating that an input port attempted to generate a signal.

C.5.2 Protected Members

The Input class has no protected members.

C.5.3 Private Members

The Input class has no private members.

C.6 The **Output** Class

The Output class is derived from the Port class and is used to simulate an output port of a component. The output ports of a component are all stored in the `O_List` public member of the Component class and can therefore be accessed by outside objects as is the case in the real world. The purpose of an Output object is to act as a gateway through which signals generated by components may be transmitted to wires connected to the output port. Because signals may be both written to and read from an output port, the Output class supports both the `get_signal()` and `send_signal()` virtual methods as defined by its parent Port class. As a result of this inheritance, the implementation of the Output class is very trivial.

C.6.1 Public Members

- `Output(Component &cmp, Connector &con, char *name)`: Like the Input constructor, the Output constructor takes a reference to the component which contains the output port, a reference to the connector that the output port feeds and a name for the output port. The connector reference and name parameters are passed to the Port base class constructor, thereby connecting the port to its external destination connector and storing the name of the port. The output port is then added to the output port list of the enclosing component.

C.6.2 Protected Members

The Output class has no protected members.

C.6.3 Private Members

The Output class has no private members.

C.7 The Runtime_Component Class

The Runtime_Component class is used to dynamically build a high-level component and all its subcomponents based upon an input stream of textual data which describes the circuit and its input signals. In the context of this project, the textual stream is sent over a command pipeline to the simulator by a graphical user interface. After construction, the circuit is simulated and its resultant outputs are sent to standard output. The GUI receives these output signals over the command pipeline and then parses and displays them in the waveform editor. The Runtime_Component is derived from the Component class and is used extensively by member functions of the Parser class.

C.7.1 Public Members

- `Runtime_Component()`: The constructor for the Runtime_Component class simply initializes its base class by invoking the Component constructor with the arguments `CKT_TIME_NULL` and with the identifying string `"Runtime"`. The `CKT_TIME_NULL` parameter signifies that the runtime component has no delay; its delay is determined by the transport delays of its subcomponents.

- `void create_input(Wire *wire, const char *name)`: This member function simply creates a new input port for the runtime component. The `wire` parameter is a pointer to the wire that feeds the input port and the `name` parameter is the unique string that identifies the port.

- `void create_output(Wire *wire, const char *name)`: This member function, like `create_input()`, creates a new output port for the runtime component. The `wire` parameter is a pointer to the wire to which the newly constructed output port sends its signals and the `name` parameter is the unique string that identifies the port.

- `void create_internal(Wire *wire)`: The runtime component may contain internal netlists which are not connected to any of its input or output ports. This constructor will add the specified wire pointer to a linked list which stores all the internal netlists contained within the runtime component.

- `Connector *find_connector(const char *name)`: This is a helper function that searches for the specified connector name amongst the input and output ports and the internal netlists of the runtime component. If the search is successful, then a pointer to the corresponding connector is returned. This method is used to determine which wires and ports are to be connected to the ports of the subcomponents of the encompassing runtime component.

- `int create_subcmp(const char *t, const char *n, List<char *>io)`: In order to create a subcomponent inside the runtime component, this method must be used. After instantiating the subcomponent, this function connects it to the specified input and output ports and to the internal netlists of the runtime component created earlier. The `t` parameter indicates what type of subcomponent to create (for example, a *NAND* gate, *XOR* gate, and so on). The `n` parameter gives the name of the component which is passed to the component's constructor. Finally, the `io` parameter is a list containing the names of the connectors to which the subcomponent is connected.

C.7.2 Protected Members

The Runtime_Component class has no protected members.

C.7.3 Private Members

- `List<Wire *> internal_netlist`: This data member is a linked list of wire pointers which stores the internal netlists of the runtime component. Internal netlists are wires which are not connected to any of the input or output ports of the runtime component. Their primary purpose is to connect subcomponents together. New internal netlists are added to the list by the `create_internal()` member function.

C.8 The Parser Class

The Parser class provides support for rudimentary parsing of textual input in stanza format. With respect to the implementation, this class is used to parse an input stream which represents a circuit description and then build a runtime component object based upon this input stream. The input stream itself is typically sent by a graphical user interface to the simulator engine. As each stanza of the input stream is read and parsed, the wires, ports and components which comprise the runtime component are constructed and connected together.

C.8.1 Public Members

• `Parser(int bufsize)`: This constructor initializes all the necessary data members of the Parser object and allocates a buffer which is used to store lines as they are read from the input stream. The number of bytes allocated for the buffer is determined by the `bufsize` parameter. By default, the buffer size is set to 512 bytes.

• `boolean ckt(Runtime_Component &cmp)`: This method acts as the main driver function which is responsible for reading the stanza headers and then invoking the correct methods for building the subentities of the runtime component. Its duties include dispatching the `read_netlist()` and `read_component()` methods, which are described next.

• `boolean read_netlist(wire_type t, Runtime_Component &cmp)`: The purpose of this member function is to parse a netlist stanza body read from standard input and store all the netlist attributes. The `t` parameter indicates whether the method is to read an `input`, `output` or `internal` stanza. After creating a new wire representing the netlist, this method will dispatch the appropriate Runtime_Component member function to create the necessary port or to add the netlist wire to the list of internal netlists maintained by the runtime component. For example, if the method was requested to read an `output` stanza, it will dispatch the `cmp.create_output()` member function. Upon encountering a `value` attribute line in an `input` stanza, the `read_netlist()` method will dispatch the `read_signals()` method.

• `boolean read_signals(Wire *wire)`: This member function will parse a list of signal values and times that follow the `values` attribute keyword in an `input` stanza. As each signal value and time is parsed, they are added to the wire parameter supplied to this method. These signals represent the primary inputs of the circuit.

• `boolean read_component(Runtime_Component &cmp)`: This member function parses the attributes in the body of a `component` stanza and constructs the desired subcomponent. Each `component` stanza body consists of three attributes. The `type` and `id` attributes are read by the member function `read_component_attribute()`, which is described next. The `port` attribute, which lists the names of the connectors attached to the ports of the subcomponent, are read and stored in a linked list of strings. Once all the subcomponent's attributes and port connector names have been identified, they are sent, as parameters, to the runtime component's `create_subcmp()` method.

• `boolean read_component_attribute(boolean &c, char *&v)`: This member function is used to read the `type` and `id` attributes of a `component` stanza body. The boolean parameter, `c`, is used to ensure that the component attribute was not already read earlier. The `v` parameter represents the actual value of the attribute. This parameter is simply a string which is dynamically allocated and assigned the appropriate value by this function.

C.8.2 Protected Members

The Parser class has no protected members.

C.8.3 Private Members

- `int get_line()`: This private member function reads in a line of text from the standard input and stores it in the buffer of the Parser object. The `line_num` data member is incremented accordingly. If the parser already had a line cached as the result of a prior call to `unget_line()`, then no line is read in from standard input — the line already in the buffer will be regarded as the current line of input.

- `void unget_line()`: During the processing of the input stream, it is sometimes necessary for the parser to put a line back in the input stream when it has read too far ahead. The `unget_line()` member function does this by leaving the line in the buffer unchanged and setting the `cached` data member to `TRUE`. The function can "unget" a line only if at least one line has already been read and the `cached` data member is `FALSE`.

- `void error(char *err)`: This member function simply displays the supplied error message, the current line number and current text line to the diagnostic log stream, `cerr`. It is used to report errors encountered during the parsing of the input stream.

- `char *get_word(const char *delim = " ")`: This function implements an elementary tokenizer for the Parser class. Upon completion, this function will return the next token in the buffer that is delimited by one of the characters in the `delim` parameter. It makes use of a temporary buffer which is dynamically allocated by this method, if necessary. The library function `strtok()` is used to extract the tokens from the current line.

- `const int line_size`: This variable represents the size of the buffer to be used by the parser to store each line of input. It is initialized by the Parser constructor. Its value should be larger than the length of the longest text line in the input stream.

- `int line_num`: This data member keeps track of the number of lines currently read. When an error is encountered in the input stream, the line number stored in this variable is displayed to aid in the debugging process.

- `char *buffer`: This data member stores the contents of the current line read from standard input. It is dynamically allocated by the Parser constructor. The `get_line()` member function actually stores the contents of the current line in the buffer.

- `char *tmp_buf`: During execution of the tokenization function, `get_word()`, a copy of the line pointed to by the `buffer` data member is made and stored in `tmp_buf`. Doing this provides the tokenizer function with a copy of the buffer to manipulate without altering the original contents pointed to by `buffer`.

- `int cached`: When a line is to be placed back into the input stream by the `unget_line()` member function, the `cached` data member is set to `TRUE`. This will inhibit a subsequent `get_line()` invocation from trying to read another line from standard input.

- `int read_one`: Upon reading a line of text from the input stream, this data member is set to `TRUE`, indicating that at least one line of standard input has been successfully read. This boolean value is consulted by the `unget_line()` and `get_word()` member functions.

C.9 The Signal Class

We can think of signals as having two attributes: a value (for example, *high*, *low* or *don't care*) and a time at which the signal occurred during the simulation. The Signal class is used to aggregate these two attributes into a single entity. Signals are frequently used by classes derived from the Connector class since these classes are responsible for the transmission of signals throughout the circuit during the simulation. Signals are also used by the lowest level components when it comes time for them to process their inputs. These components take their input signals from their input ports and produce appropriate signals which are sent to their output ports.

C.9.1 Public Members

- `Signal(Sig_Val val, ckt_time time)`: The Signal constructor accepts a signal value (which is simply an enumerated type) and the time at which the signal occurred. The corresponding private members of the Signal object are then initialized with these values.

- `friend ostream &operator <<(ostream &os, const Signal &sig)`: This is an overloaded operator function which lets the implementation use the standard insertion operator (`<<`) to display the value and time of a Signal object on standard output.

- `Sig_Val get_value()`: This is a simple accessor function which returns the value of the signal.

- `ckt_time get_time()`. This is a simple accessor function which returns the time at which the signal occurred.

• `void set_value(Sig_Val newval)`: This method changes the value of the signal to the specified value. It is used only by the `replace()` method of the Wire class and is intended for future support of zero-delay component simulation.

C.9.2 Protected Members

The Signal class has no protected members.

C.9.3 Private Members

• `Sig_Val value`: This is an enumerated type which contains the value of the signal. This value can be one of `SIG_HIGH`, `SIG_LOW`, `SIG_X` or `SIG_NULL`. The latter signal value represents an invalid signal.

• `ckt_time t`: This data member represents the time at which the signal occurred. The `ckt_time` type is defined to be of type `long` in this particular implementation. Two constants, `CKT_TIME_INIT` and `CKT_TIME_NULL` represent an initial time and an invalid time respectively.

C.10 The List Class

The List class provides generic support for linked lists. The template mechanism of C++ is used to make the generic list typesafe. Auxiliary classes List_Node and List_Iterator are used to store the generic data associated with each node of the list and to support iteration over the elements of a linked list, respectively. The implementation of these auxiliary classes is trivial; they are therefore not discussed in this section.

C.10.1 Public Members

- `List()`: The List constructor initializes the head and tail of the linked list to the null pointer and initializes the number of elements currently in the list to zero.

- `add(const Type &item)`: This member function adds the element specified by `item` to the linked list. Memory for the new item is allocated from the free store and the element is appended to the end of the linked list. The head and tail of the linked list are updated accordingly and the internal counter maintaining the number of elements in the linked list is incremented by one.

- `int num_elements()`: This member function returns a count of the number of elements currently in the linked list. Because the List class maintains a count of the number of list elements as new items are added, this member function operates in constant time.

- `int is_empty()`: If the linked list contains no elements, this member function will return non-zero; otherwise, it will return zero.

- `const Type *last_element()`: Occasionally, an implementation may wish to obtain the last item in a list. The `last_element()` member function will return a pointer to the last element of the list, or a null pointer if the list is empty. This function operates in constant time since the List class maintains a pointer to the last member of the list.

C.10.2 Protected Members

The List class has no protected members.

C.10.3 Private Members

- `List_Node<Type> *head, *tail`: These two data members store pointers to the first and last elements of the list respectively. They are updated as new items are added to the list.

- `int count`: This data member maintains a count of the number of elements currently in the list. Its value may be accessed by the `num_elements()` member function.