PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Y5207 .A61 .P73 .N4

Charles, Not a reference

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

ROOMMS
A RELAXATION-BASED, OBJECT-ORIENTED, MIXED-MODE
SIMULATOR

by

JEFFREY ALLAN WATTS, B. Eng., P. Eng.

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of**

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering

**Department of Electronics
Carleton University
Ottawa, Ontario
18 May 1993**

©

1993, Jeffrey Allan Watts

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Yukon - Bibliothèque

Chorley - Bibliothèque

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-84125-7

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,

**ROOMMS, A RELAXATION-BASED, OBJECT-ORIENTED,
MIXED-MODE SIMULATOR**

submitted by
Jeffrey Allan Watts, B.Eng., P.Eng.

in partial fulfilment of the requirements
for the degree of Master of Engineering.

Chair, Department of Electronics

Thesis Supervisor

Carleton University

May 1993

Abstract

The increased capability of a circuit designer to put more functionality on an integrated circuit is driving the need for more and more capable circuit simulators. A simulator model was developed to visualize the organization of a simulator and the theory behind simulators was investigated in order to visualize an ideal simulator. A mixed-mode simulator was considered as best to meet the ideal simulator requirements. An object-oriented development environment (Objective C) was chosen and a working mixed-mode simulator, ROOMMS, was developed. Results of analog and mixed-mode simulations of an in-service telecommunications circuit are presented. ROOMMS implements the Iterated Timing Analysis algorithm in an analog mode and a proven logic gate mode and a set of behavioural-level sources. ROOMMS was shown to readily allow the incorporation of new behavioural/logic/analog element models and modifications or additions to the simulation algorithms.

Acknowledgements

I would like to thank Dr. Tadeusz Kwasniewski, my supervisor, for his guidance and friendship during my studies. His support and patience have been much appreciated.

I would like to thank Dr. Miles Copeland for his helping and encouraging me to follow graduate studies. His initial kick, and his continued support resulted in this work.

I would like to thank the academic and support staff at the Department of Electronics, Carleton University for their constant help, support and sympathy.

I thank my employers, the Department of National Defence and my office, the Project Management Office, Tactical Command, Control and Communications System for their support and patience during my studies. I would especially like to thank Colonel Humphrey Sheridan (Ret'd), Lieutenant Colonel John O'Halloran and Mr. Brent Marsh for their support and encouragement.

Lastly but not least, I would like to thank my parents and family. I could not have done this work without Mom and Dad.

Table of Contents

Chapter 1	Introduction	1
1.1	Thesis Motivation and Objectives	1
1.2	Thesis Outline	4
Chapter 2	Mixed-Mode Simulation	5
2.1	Introduction	5
2.1.1	Simulation	6
2.1.2	Simulator Model	9
2.1.3	Simulator Levels or Modes	12
2.2	Behavioural Modelling	13
2.3	Data Path Modelling	13
2.4	Intermediate Level Simulation	14
2.4.1	Methodologies	14
2.4.2	Element Models	15
2.5	Switch-Level Modelling	15
2.6	Timing Simulation	16
2.6.1	Methodologies	16
2.7	Analog Simulation	17
2.7.1	Methodologies	17
2.7.2	Equation Derivations	18
2.7.2.1	Circuit Coupling	24
2.7.2.2	Timestep Interval	25
2.7.3	Algorithms	26
2.7.3.1	Modified Nodal Analysis	26
2.7.3.2	Relaxation Methods	27
2.7.3.2.1	Iterated Timing Analysis	27
2.7.3.2.2	Waveform Relaxation	28
2.7.4	Element Models	29
2.8	Integrated Circuit Layout Characterization	31
2.9	Mixed-Mode Simulators	32
2.9.1	SPICE	33
2.9.2	SPLICE1.7	34
2.9.3	iSPLICE3	34
2.9.4	ELDO	35
2.10	Conclusions	36
Chapter 3	ROOMMS - A Relaxation-based, Object-Oriented, Mixed-Mode Simulator	39
3.1	Introduction	39
3.2	Algorithms	40
3.2.1	Logic Simulator Algorithms	40
3.2.1.1	Logic Simulator Delay Modelling	43

3.2.2	Analog Simulator Algorithm	44
3.3	Device Models	46
3.3.1	Analog Device Models	46
3.3.2	Logic Device Models	47
3.3.3	Source Elements	47
3.3.4	Grounded Capacitor	48
3.4	Implementation of ROOMMS	49
3.4.1	Program Organization	49
3.4.2	Implementation History	55
3.5	Adding Functionality to ROOMMS	60
3.6	Summary	61
Chapter 4	Testing Results from ROOMMS	63
Chapter 5	Conclusions	71
5.1	Summary	71
5.2	Achievements and Contributions	72
5.3	Suggestions for Further Work	73
	References and Bibliography	75
	Appendix A Software Development Environment	79
	Appendix B ROOMMS Class Descriptions	82
	Appendix C ROOMMS User Guide	133
	Appendix D An Example ROOMMS Input Deck	159
	Appendix E Device Model Equations	165

List of Tables

Table 1 Relative Simulator Costs	32
--	----

List of Figures

Figure 2-1 Simple Circuit Problem	6
Figure 2-2 Simple Circuit Problem Example Source Deck	8
Figure 2-3 Simulator Model	10
Figure 3-1 Logic Node Processing Algorithm	41
Figure 3-2 Logic Node Scheduling Algorithm	42
Figure 3-3 Iterated Timing Analysis Algorithm	45
Figure 3-4 Simplified Class Hierarchy	49
Figure 3-5 Elements Class Hierarchy	51
Figure 3-6 Sources Class Hierarchy	52
Figure 3-7 Nodes Class Hierarchy	53
Figure 3-8 ROOMMS Class Hierarchy	54
Figure 3-9 ROOMMS Class Hierarchy Development	57
Figure 4-1 Boot-strapped Inverter	64
Figure 4-2 Boot-strapped Inverter Output	64
Figure 4-3 Mixed-Mode Circuit	65
Figure 4-4 Mixed-Mode Circuit Output	66
Figure 4-5 Folded Cascode Amplifier of Filter	67
Figure 4-6 Switched-Capacitor Pulse Shaping Filter	68
Figure 4-7 Transmit Filter Output	70

List of Terms

ALU	Arithmetic Logic Unit
AnalogElements	ROOMMS Class - superclass of analog element models
AnalogNodes	ROOMMS Class - class definition of analog nodes
Andgate1	Sample ROOMMS model
bjt	bipolar junction transistor
"C"	the programming language
calculateNewOutput	ROOMMS method - calculations for analog element models
ci	input capacitance of logic gate
CircuitInput	ROOMMS Class - input routine
co	output capacitance of logic gate
Cmodel	Sample ROOMMS element model label
currentVoltage	ROOMMS variable
deltaV	ROOMMS variable
ELDO	name of a simulator
ELogic	simulator mode - Electrical Logic
falldelay	calculated delay of a signal on a logic node (0 to 1 transition)
fanin	property of a node - element which has the node as an input node
fanout	property of a node - element which has the node as an output node
Fcapr	ROOMMS Class - floating capacitor
Gcapr	ROOMMS Class - grounded capacitor
Handle Timestep	Process from simulation model
HSPICE	commercial variant of SPICE
I/O	Input/Output
IOD	Input Output Device
Isourc element	ROOMMS Class - Current source
iSPLICE3	version of SPLICE
ITA	Iterated Timing Analysis
lasttime	ROOMMS variable
Lcvcv	ROOMMS Class - Logic Clocked Voltage Controlled, Voltage Source
Lcv	ROOMMS Class - Logic Controlled Voltage Source
LogicElements	ROOMMS Class - superclass of logic element models
LogicNodes	ROOMMS Class - class definition of logic nodes
LogicState	ROOMMS Class - class containing data structure consisting of logic value and logic strength
Lrail	ROOMMS Class - Logic Rail
LSI	Large Scale Integration
Lsrc	ROOMMS Class - Logic source

LU	Lower/Upper triangular decomposition (of matrices)
METHODS	Object-Oriented Programming Environment - version of SMALLTALK
MOS	Metal-Oxide-Semiconductor
MOSFET	Metal-Oxide Semiconductor Field Effect Transistor
Mrt	Minimum Resolvable Timestep
newElementOfClass	ROOMMS method - creates a circuit element's data structure during setup phase
nextTime	ROOMMS variable
presentTime	ROOMMS variable
previousGuess	ROOMMS variable
previousVoltage	ROOMMS variable
PSPICE	commercial variant of SPICE
RATFOR	RATional FORtran - programming language
risedelay	calculated delay of a signal on a logic node (1 to 0 transition)
Rmodel	Sample ROOMMS model
ROOMMS	Relaxation-based, Object-Oriented Mixed-Mode Simulator
RTL	Register-Transfer-Logic
SAN	ROOMMS Class - Source Analog Node
SimulationUtilities	ROOMMS Class - utility subclass
SLN	ROOMMS Class - Source Logic Node
SMALLTALK	object-oriented programming language/environment
SOR	Successive Over-Relaxation Parameter - a simulation parameter
SpecialDependencyObject	ROOMMS Class - utility superclass
SPICE	name of a simulator
SPLICE1.7	name of a simulator
STR	"C" string
tcq	propagation delay between clock, set/reset and Q for flip flops
tdcs	set-up time for input and rising edge of clock for flip flops
tf	fall delay of gate (1 to 0 transition)
tfc	falling driving strength of gate - used during delay calculation for logic node
timestep	discrete time interval during simulation
Timestepper	Process from simulation model
totalCapacitance	ROOMMS variable
totalConductance	ROOMMS variable
totalCurrent	ROOMMS variable
tpw	clock pulse width plus hold time for flip flops
tr	rise delay of gate (0 to 1 transition)
trc	rising driving strength of gate - used during delay calculation for logic node

Tres	ROOMMS Class - resistor
Tsrc	ROOMMS Class - Timing source
Vch	ROOMMS Class - Voltage controlled logic source
Vevs	ROOMMS Class - Voltage controlled voltage source
VLSI	Very Large Scale Integration
Vmodel	Sample ROOMMS element model label
voltageChange	ROOMMS variable
Vrail	ROOMMS Class - Voltage Rail

Chapter 1 Introduction

1.1 Thesis Motivation and Objectives

Advances in the expertise needed to put more functionality into smaller packages have created a growing need for more complex devices. Conversely, the demand for more complex devices has created a growing need for advances in the expertise needed to put more functionality into smaller packages. These two trends create a tight feedback loop.

Society's reliance on more and more complex electronic packages in everything from computers and telecommunication devices to consumer items such as microwave ovens has created a competitive market. To satisfy this market, more functionally complex packages must be implemented quickly using a minimum of costly hardware implementation prototypes. Design and construction errors must be removed from these packages before they are built into costly hardware.

This thesis will look at ways of simulating those packages which are implemented as integrated circuits. Design and construction faults can be reduced by simulating the integrated circuits before implementing hardware. Tools are needed and have been created to do this simulation, tools that support the systematic design of increasingly complex circuits [1,2,3,4]. The hardware advances resulting in more functionality in integrated circuit packages have required matching advances with simulator tools.

In the 1970s, electrical and timing simulators were available to prove circuit designs at the discrete device level [5] using models of circuit elements such as transistors and capacitors. Still, even an LSI circuit of 10 000 devices would be expensive to simulate at the electrical and timing levels. More abstract ideas could be simulated using switch or gate level simulators [5], but the greatest cost saving would be to isolate analog device detail from the design behaviour of the circuit. This separation into analog and digital domains reduces the need to simulate a circuit at the electrical and most expensive level. Digital domains reduce a design's possible faults by reducing the sensitivity of the circuit design with respect to analog circuit details. By abstracting the design from electrical circuit characteristics, the simulation costs are reduced, since more abstract signal quantities can be used, and dependence on costly detailed device models is reduced, also reducing simulation overhead costs. Circuit blocks, starting with simple logic gates and becoming increasingly more complex to reach microprocessor "cells" have been thoroughly tested and introduced as gates or behavioural blocks for more advanced simulators [4].

Eventually, circuit densities and the need and ability to mix analog and digital designs on a single integrated circuit (for telecommunication circuitry, for example) have brought the simulation problem back to how to simulate large complex designs when designs are sensitive to design defects at the lowest level of detail. The requirement then is for a simulator which can test parts of a circuit that have the potential for design faults at an electrical level, while simulating other parts of a circuit which are not sensitive to

low level faults at higher levels of abstraction so that simulation costs are reduced. Such a simulator is called a mixed-mode simulator.

Consider the transmit filter portion of a telecommunication transceiver chip which must filter a coded signal to remove unwanted spectral content from the input signal before transmission [6]. This circuit could take the form of a finite-impulse response filter. Such a circuit would sample its input signal, manipulate that signal digitally, and then convert the signal back into an analog form minus the unwanted high frequency component. The circuit designer will want to simulate this circuit, including the critical sampling, digital manipulating and analog signal reconstruction functions. Since the circuit designer will be using a well proven set of digital circuit components, he or she will not be interested in simulating the digital parts of the circuit at an electronic level. Therefore the circuit designer will want to simulate this circuit at two levels, at an electrical or analog level for the sampling function, at a digital level for the digital manipulating function, and again at an analog level for the signal reconstruction portion of the circuit. This thesis will show how the use of a mixed-mode simulator will support the circuit designer in proving the design of his or her filter design.

The primary objective of this thesis is to explore the concept of simulators and what an ideal simulator should be in order to meet the needs of the telecommunication and other circuit design communities, to create such a simulator, and to discuss the development of this simulator and demonstrate its functionality. A secondary objective is to explore the relatively new concept of object-oriented design and programming, and how this can allow the development of a simulator within the constraints of this project.

A final objective is to explore how the object-oriented design and programming environment eases future modifications.

1.2 Thesis Outline

Chapter 2 begins with an introduction to circuit simulation. A simulation model is presented as a reference model, and simulation levels, algorithms, and device models are described. The results of a review of existing simulators are presented, and an ideal simulator is discussed. A new simulator, ROOMMS, is presented in Chapter 3. Algorithms, device models and implementation details are provided. Chapter 4 presents simulator test results including the functionality test of a portion of a telecommunication circuit simulated using ROOMMS. Chapter 5 provides a conclusion followed by suggestions for further work.

Chapter 2 Mixed-Mode Simulation

2.1 Introduction

A circuit is designed using a mixed approach of top-down "black box" design and a bottom-up approach using existing circuit elements. These circuit elements may range from discrete devices, such as transistors, to gates or other more complex blocks of many tens, hundreds or even thousands of transistors, all of which have been proven or characterized at some earlier date. Thus at any given point in the design process, the circuit design could be summarized as containing multiple levels of detail ranging from high level "black boxes" (which are expected to do something specific, though how they do so is not yet defined), to pre-designed circuit components (which may be characterized as a whole or defined to be a previously designed subcircuit), to discrete devices used as interconnections between "black boxes". Regardless of detail, the designer is most interested in knowing that the circuit works as presently designed. How can the circuit be simulated? The designer may simulate the behaviour of the circuit as conceived at a high level, ignoring the more detailed aspects of design, or the designer may simulate the parts of the circuit that have been subject to more detailed design independent of the rest of the circuit, thereby losing information about how the circuit works together. The designer wants to be able to simulate the design as it stands, at multiple levels of detail so that the whole of the current design can be tested. This leads to the requirement for a simulator that can handle multiple levels of detail.

2.1.1 Simulation

The functional verification of integrated circuits is accomplished by applying mesh or nodal analysis techniques to a circuit to solve for currents and/or voltages at all points in the circuit for some simulation period [7,8]. In anything but the most simple circuits and signal waveforms, automation is required for this verification. The automated analysis of circuits, or simulation, requires the adoption of a nomenclature which will be introduced here.

The RC circuit of Figure 2-1 contains three circuit elements: a voltage source; a

Figure 2-1
Simple Circuit Problem

resistor; and a capacitor. These elements are connected together at three circuit nodes: a ; b ; and c . If the elements in this circuit are assumed to be ideal, then the signal values on each node of this circuit can be worked out by hand, based on the input signal v , the resistance, R , and capacitance, C . When the circuit becomes more complex, and especially if non-linear element models are used, the simple analysis becomes too tedious

to do by hand. In addition, hand calculations of more complex circuits have an inherent prospect for error.

The margin for error in analysing more complex circuits is reduced markedly by automating the analysis. A set of equations is created for each node, of the form $v = Gi$, where G is sum of the conductance of the elements attached to the node, v is the voltage on the node and i is the sum of the current entering the node. These equations are solved for a value of i based on the existing voltages in the circuits. This value of i can be then used to calculate a voltage on the node at time t . The behaviour of the circuit is then determined by doing this analysis at discrete time intervals over the length of time in question. The discrete time intervals or timesteps can be selected so that the equations required to model a non-linear element, such as a capacitor, can be linearized. For Figure 2-1, the capacitor can be modelled using the equation [7,8]:

$$i = \frac{C}{h}(\Delta v_b - \Delta v_c) \quad (1)$$

where h is a discrete timestep or time period,

$$\Delta v_b = v_b(t) - v_b(t - h), \text{ and}$$

$$\Delta v_c = v_c(t) - v_c(t - h).$$

An analysis of this type requires previous knowledge of the voltage values on nodes in the circuit. Therefore, some process is required to keep the analysis consistent in terms of the simulation time so that the circuit is simulated correctly. The controlling

process is called a scheduler, which ensures that the analysis of a node takes place at discrete timesteps in the correct order of time.

The task of automating the analysis consists of putting the circuit description into the proper form for analysis, and then obtaining the results, again in some meaningful form. In order to simulate the circuit, the correct forms of the circuit node equations introduced above must be entered into the simulation engine. This is accomplished by describing the circuit, the element models used to model the elements in the circuit, and information required to control the analysis. One way would be to enter the circuit description as a schematic diagram as in Figure 2-1. The element models would be

Simple Circuit Example Source Deck

Model Cards

```
.model R:model tres: 10000
.model Cmodel fcapr: 10e-12
.model vmodel sinusoid: 2.5 0.0 0 5.0e-8
```

Element Cards

```
.element v-element a-node c-node vmodel
.element R-element a-node b-node Rmodel
.element C-element b-node c-node Cmodel
```

Simulation Parameters

```
.time 5.0e-9 150.0e-9
```

Output Requests

```
a-node b-node c-node
```

Figure 2-2
Simple Circuit Problem
Example Source Deck

assumed to be previously defined, and any parameters of those elements would be entered

in some other manner. The information required to control the analysis could also be entered in some other manner, such as creating a textual circuit description. Such a description, called a source deck, is shown in Figure 2-2 [9,10]. This figure shows the information required for an analysis. The source deck is begun with a title to identify the analysis. The element's individual characteristics are then defined in model cards which are parameters passed to predefined element models. The circuit's interconnections are then described in element cards which define the relationships between the elements in the circuit. Next, simulation parameters and information required for the analysis are listed - the minimum time interval for the analysis and the duration of the analysis. Finally the method of showing the results of the simulation is listed. These results could be in the form of listings of voltage values, graphical displays of signals, or some other form of information. Details of the syntax shown in Figure 2-2 are contained in Appendix C.

2.1.2 Simulator Model

The main processes of a simulation engine, a simulator, are shown in Figure 2-3. This conceptual model shows the main processes of the simulator, the Scheduler, Node and Element processes, the controlling Timestepper and Handle Timestep processes, and the main data flows between these processes.

This model fits any simulator which uses a scheduler or event-wheel to schedule the processing of nodes. It assumes that a data structure representing the circuit under

Figure 2-3
Simulator Model

simulation has been created from some form of input, and that the simulator is ready to run. All preprocessing detail has already occurred and the simulator model shows only the simulation phase of the simulator.

The simulation is started, and the Timestepper process obtains each timestep in turn until the simulation is done, and passes it to the Handle Timestep process, which does further processing. The Timestepper process waits for Handle Timestep to finish

processing each timestep before it obtains the next one from the Scheduler process. The Scheduler process is responsible for dispensing and acquiring information from and to the Scheduler List data structure. The Scheduler process returns the next timestep in the Scheduler List to the Timestepper process (Note that the Scheduler List is not necessarily continuous). The Scheduler process returns the next node (or event) in a particular timestep list. It also receives nodes from the Node process, which are then scheduled as determined by the Node process.

The Handle Timestep process receives the present timestep from the Timestepper process, and proceeds to obtain each node in that timestep's list from the Scheduler List. Handle Timestep tells the Node process that each node passed to Handle Timestep by the Scheduler process is an event. The Handle Timestep process waits until the Node process has finished dealing with the event before passing a new event (received from the Scheduler process) to the Node process. This continues until the end of the present timestep's list.

The Node process accepts each event from the Handle Timestep process. The Node process determines a new state on the node by telling the Element process to calculate a new state on the output node of each of the node's fanin elements. The Node process resolves any conflicts involved if the node has more than one fanin element, and then determines if the state on the node has changed. If the state has changed, the Scheduler process is told to schedule each of the node's fanout element's output nodes, either at the present timestep, or after some delay, depending on the model parameters of the fanout elements. If the state on the node does not change, the node is allowed to

relax, i.e. no changes are to be propagated so no nodes are scheduled. The Node process determines if the current event is an output node, and if so, outputs the state of the node. The Node process then finishes with the current event.

The Element process uses a representative model or algorithm of the actions of a particular element to calculate the new state on the output node of an element. Such "built-in" models determine the new output state based on the element type, the element's input nodes' states, and the element's model parameters. The Element process receives the element from the Node process and returns the new output state to the Node process.

This simulator model (Figure 2-3) shows that implementation details for key aspects of the simulator can be compartmentalized quite simply. Node processing details are contained within the Node process; scheduling details are contained in the Scheduler process; element model behaviour is contained within the Element process. There is little cross-over, and implementation details can remain somewhat independent of each other. This modularity eases the transition from conceptual ideas to code, especially in an object-oriented programming environment, since one of that environment's main features is encapsulation, where implementation of a class occurs independently of its interface to the outside.

2.1.3 Simulator Levels or Modes

A multiple level simulator (alternatively, a multi-level, multi-mode, or mixed mode simulator) integrates some, parts of, or all of the following: behavioural modelling; data

path modelling; gate or intermediate-level modelling; switch-level modelling; timing behaviour of discrete device circuits; detail device level behaviour or analog simulation; and physical layout modelling [3,5,8,11]. The mix of these levels of simulation detail should reflect the levels of design detail seen previously.

2.2 Behavioural Modelling

Behavioural modelling consists of testing whether the algorithms or premises of the overall circuit and its major subcircuit blocks (the "black boxes") are correct. Behavioural modelling can prove that the expected outputs of a circuit or circuit block will occur, given a representative range of inputs. Behavioural modelling can also test whether the interconnections between circuit blocks are correct [4].

2.3 Data Path Modelling

Data path modelling consists of testing that data is passed correctly at slightly more detailed levels of design. As an example, consider a microprocessor design using an Arithmetic Logic Unit (ALU), microcode stores, register stores, and Input/Output (I/O) circuitry as circuit blocks. Data path modelling can be used to simulate the movement of data from the I/O block to the register store, and the information or data passed between the ALU, the microcode store and the register store as the data in the register

store is acted upon. Data path modelling is also known as Register-Transfer-Level (RTL) modelling [4].

2.4 Intermediate Level Simulation

Gate or intermediate level modelling consists of modelling a circuit consisting of logic gates or analog gates of equivalent complexity [2,5,12]. In logic gate simulation, an abstract signal domain, consisting of high, low and unknown signals, and driving strengths, requires less computational overhead than more detailed forms of simulation, while providing detailed circuit behaviour results.

2.4.1 Methodologies

One example of intermediate level modelling [2,5,12] is a logic gate simulator. Such a simulator models discrete signal levels: 0 or low, 1 or high, and unknown. The ability of a real circuit to drive a load at its output can be represented by driving strengths. A very weak strength represents a tri-state logic signal. A higher strength can drive a heavier load or dominate the output of a gate with a lower strength. In most circuits most nodes will change infrequently, while a smaller number of other nodes will change more often. Those nodes which are not changing at any particular time are said to be relaxed [2,5,12]. When a node is not changing, it is not necessary to determine a new value for that node. A scheduler can be used to determine whether a node is

changing, and whether the node should be evaluated or not. The significant problem in logic simulation is one of scheduling the nodes that are changing without introducing transients, called glitches, which are spurious manifestations of the scheduling algorithm. In a real circuit, nodes change value simultaneously in response to stimuli. In a simulation, nodes are evaluated one after the other and iteratively, and until the value of each node in the whole circuit is determined, a change from any one node evaluation may be temporarily incorrect. If that incorrect change is propagated by the scheduler, a glitch is said to have occurred.

2.4.2 Element Models

The element models of a logic simulator determine a logic state on their outputs based on the logic state of their inputs. Simple combinatorial gates can be modelled using look-up tables. Sequential elements require a memory of previous states. Gate element models should also be capable of providing propagation delays through the element, load driving capabilities, and input and output node capacitive loading [2,5,12].

2.5 Switch-Level Modelling

Switch-level modelling consists of modelling the behaviour of discrete elements such as transistors using simplified models of those elements. Switch-level modelling can use discrete signals such as logic levels or continuous signals such as voltage or current.

For example, a transistor can be modelled as an ideal switch. Circuit behaviour can be simulated using this approach, but detail is lost in terms of delay modelling [8,13]

2.6 Timing Simulation

Timing behaviour of discrete device circuits can be modelled using models of transistors, resistors, capacitors, and other circuit components, and a simplified algorithm that can give first-order approximations of the timing of the circuit [5].

2.6.1 Methodologies

One example of timing simulation is the switch-level timing simulation, ELogic. ELogic is a simulation technique which uses analog element models and discrete signal voltage values [13,14,15]. Signals are propagated through elements based on a signal change from one discrete voltage level to another in some determined time delay. Since the changes in the signals in the circuit under test are discrete, they change less often, allowing more circuit relaxation. This means that an ELogic simulation requires less computation time, while keeping much of the timing and signal behaviour of the full analog simulation.

2.7 Analog Simulation

Detailed device behaviour modelling, the most detailed level of element modelling, is called electrical or analog simulation. At this level of simulation, the behaviour of circuit devices is represented as accurately as possible using discrete elements such as transistors, capacitors and resistors. The more detailed the element model, the more computationally intensive the task [2,3,4,9,11].

2.7.1 Methodologies

Simulating a circuit involves describing the circuit's internal signals, or voltages and currents on circuit nodes, in the form of a set of equations, and then solving these equations for values of the signals at any particular time of interest. Two methods are of interest in solving these equations. The first involves direct matrix solution of the set of equations to get the unknown signal values; the second involves iteratively solving the equations by guessing values for all but one signal, finding the corresponding value for that signal, and using that new value as a guess to solve for another signal value. This is repeated until all the signals are resolved for the circuit in a repetitive, iterative manner. Of the two methods, the first is more exact, and more likely to converge mathematically to a solution [4]. The second can be mathematically unstable, and therefore is less accurate and less dependable [4]. However, the second does have advantages and these will be shown in the following paragraphs.

Simulation solution involves solving all the circuit's node equations to determine the signals in the circuit. It can be shown that any system and its dynamics can be represented by a state variable equations [4,8,10] of the following form, which describe the system's states, derivatives, and input conditions:

$$\mathbf{P}(\dot{\mathbf{x}}(t), \mathbf{x}(t), \mathbf{u}(t)) = \mathbf{0} , \quad \mathbf{x}(0) = \mathbf{X} \quad (2)$$

where $\mathbf{x}(t)$ is the vector of unknown signals,

$\mathbf{u}(t)$ is a vector of independent sources (voltage, current, charge, etc.), and

$\mathbf{x}(0) = \mathbf{X}$, the initial conditions.

The vector $\mathbf{x}(t)$ may include any of the following: node voltages, branch currents, capacitive charges, and inductive fluxes.

Equations of the form of equation (2) are created by applying basic circuit analysis techniques, including Kirchhoff's Current Law, Kirchhoff's Voltage Law, and branch constitutive equations, which are formulations of the admittance between the nodes (i.e. on the branches) of the circuit.

2.7.2 Equation Derivations

This section will derive equations from the general state variable equation (equation (2)) which can be used to calculate the behaviour of a circuit.

Equation (2) can be rewritten as [4,8,10]:

$$YV = J \quad (3)$$

where Y is the admittance matrix, V represents the node voltages, and J is the current flowing into the nodes from some reference node. This equation can also be expressed as:

$$\begin{bmatrix} y_{11} & y_{12} & \dots & y_{1,n+1} \\ y_{21} & y_{22} & \dots & y_{2,n+1} \\ \dots & \dots & \dots & \dots \\ y_{n+1,1} & y_{n+1,2} & \dots & y_{n+1,n+1} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \\ \dots \\ V_{n+1} \end{bmatrix} = \begin{bmatrix} J_1 \\ J_2 \\ \dots \\ J_{n+1} \end{bmatrix} \quad (4)$$

where there are n circuit elements, and consequently $n + 1$ circuit nodes (with an $n + 2$ nd reference node, ground, or node 0). The admittance y_j for each element in the circuit can be determined by connecting all nodes, except the j th to the reference ground, and applying a voltage source to the j th node. This determines:

$$y_{jj} = \frac{J_j}{V_j} \quad \left| \text{all nodes except the } j\text{th grounded} \right. \quad (5)$$

which is the j th column of the matrix Y .

It can be shown that it is possible to determine the terms of the matrix Y by inspection [7]. For a circuit with n circuit elements and therefore $n+1$ nodes, the admittance matrix terms are determined by the following rules [7]:

- i. The diagonal entries of Y are positive. y_{jj} are given as:

$$y_{jj} = \sum \text{admittances connected to node } j \quad (6)$$

- ii. The off-diagonal entries of Y are negative. y_{jk} are given as:

$$y_{jk} = \sum \text{admittances connected between nodes } j \text{ and } k \quad (7)$$

- iii. The j th entry of the matrix J in Equation (4) is given as:

$$J_j = \sum \text{currents from independent sources entering node } j \quad (8)$$

Equations (2) and (3) are normally non-linear. As a rule, it is considered that non-linear equations cannot be solved in a closed form [16]. Numerical techniques are used instead to solve the equations just developed (equations (3)-(7)). Because of the time derivatives present in equation (2), it is necessary to apply numerical integration

techniques to solve the equations moment by moment over the life of the simulation. In numerical integration, time is divided into discrete time intervals:

$$t_{i+h} = t_i + h, \quad 0 \leq i \leq \frac{T}{h} \quad (9)$$

where t_i is the i th discrete time interval or i th timestep

h is the time interval or timestep

T is the total length of time under simulation.

The selection of the timestep interval h will be discussed further in Section 2.7.2.2.

A first-order implicit integration technique, called the Backward-Euler method [7] is chosen to solve a form of equation (2):

$$\dot{x}(t) = f(x(t)) \quad (10)$$

using an equation of the form:

$$x(t_{i+h}) = x(t_i) + hf(x(t_{i+h})) \quad (11)$$

The numerical integration technique converts the set of non-linear differential equations into a set of non-linear algebraic equations that must be solved at each timestep. The set of non-linear algebraic equations can be solved under favourable conditions using the Newton-Raphson Method [16]. In a circuit with $M=n+1$ circuit nodes (equation (4)), there are M equations to be solved with M unknowns. For the variables $x_i, i = 1, 2, \dots, M$:

$$f_i(x_1, x_2, \dots, x_M) = 0 \quad (12)$$

The Newton-Raphson technique involves estimating the value of the function at time $t + h$ using the equation:

$$x_i(t+h) = x_i(t) + hf_g(x_g(t+h)) \quad (13)$$

where $f_g(x_g(t+h))$ is a guess of the value of $f(x(t+h))$.

The first guess at a particular time can be determined from past events. A first-order guess can be determined from:

$$x_g(t+h) = x_i(t) + \{x_i(t) - x_i(t-h)\} \quad (14)$$

Then the value of $x_i(t+h)$ is determined by solving the system of algebraic equations of equation (13). If the values of $x_i(t+h)$ and $x_g(t+h)$ are within some determined tolerance value, then the equation is considered solved. If the tolerance ϵ , where

$$\epsilon \geq |x_g(t+h) - x_i(t+h)| \quad (15)$$

is exceeded, then the guess is re-assigned so that

$$x_g(t+h) = x_i(t+h) \quad (16)$$

and equation (13) is solved again. The process continues until the difference between the calculated value and the guess is within the tolerance ϵ .

The Newton-Raphson Method of solving a set of non-linear algebraic equations works under favourable conditions. A description of these favourable conditions follow. The values of $x_i(t)$ can vary greatly between any time t and time $t+h$, or they can change little. It can be shown that the dependency of a change of one element of the vector described by $x_i(t)$ to a change of another element of that vector is a function of how coupled the circuit is [3]. The more interconnections of circuit nodes, the more complex and interdependent the set of circuit node equations. When that set of equations is expressed in matrix form, the matrix is seen to be very full or dense. Therefore the more interconnected or coupled a circuit is, the more dense the equation matrix, and the greater the dependency of a change of one element of the vector $x_i(t)$ to a change of another element of that vector. Such a circuit is said to be "tightly coupled" [3,15]. This has the effect that the guess (equation (14)) will be further from the ideal (since the guess is determined from the behaviour of individual elements of the vector $x_i(t)$ and ignores the interdependencies between the elements of the vector) requiring more iterations to reach a solution. The further from the ideal solution, the lower the likelihood that a solution will be reached in any reasonable number of iterations.

A circuit with a lesser degree of coupling will produce a sparser matrix [3]. The sparser the matrix, the fewer the dependencies among the elements of $x_i(t)$ (since many elements are zero), and therefore the better the guess from equation (14) will be. The better the guess, the fewer iterations required, and the more favourable it is that the Newton-Raphson Method will work.

The second variable in equation (14) that affects how good a guess has been made is the time differential, or timestep, between t and $t+h$. The smaller the timestep, h , the less that $x_i(t+h)$ will vary from $x_i(t)$. Therefore, the smaller the timestep, the more favourable the conditions for the Newton-Raphson Method.

Two ways which increase the likelihood of obtaining the solution using the Newton-Raphson Method have been shown: reducing the coupling of the circuit network [3,12]; and selecting a small enough timestep [3,4,7,11]. These will be explored in the following sections.

2.7.2.1 Circuit Coupling

When circuits are closely coupled, the value of $x_i(t)$ changes rapidly between timesteps [3]. The characteristic matrix of a closely coupled circuit as generated by equation (4) is dominated by none-zero matrix nodes. As the timestep is reduced, the effect of each node in the matrix is reduced since the significant changes on each node are reduced [3].

Another way of reducing the coupling is to add grounded capacitance elements to each node [3,12]. Adding capacitances that do not explicitly exist to a circuit may seem bizarre, but each circuit node has in fact a small but significant parasitic capacitance to ground. These small capacitances, which the circuit designer does not always consider, still have an effect on the circuit, and explicitly defining these capacitors greatly reduces the coupling in a circuit.

2.7.2.2 Timestep Interval

When the value of $x_i(t)$ varies greatly between time t and $t+h$, the guess produced by equation (14) has the highest potential of being wrong because the equations for the element are not likely to be linear over a large change. However, if the change in $x_i(t)$ is small, the guess produced by equation (14) is more likely to be correct. If the difference between $x_i(t+h)$ and $x_g(t+h)$ is small, then it is possible that this difference produced with a slightly larger timestep h would also be small, meaning that the timestep h could be increased without significantly affecting the guess.

Increasing the timestep size would have obvious benefits in terms of processing time savings during the simulation. One drawback of a varying timestep is the added computations required to determine the best timestep. A second drawback is the possibility that a chosen timestep will make the solution impossible, which will require reducing the timestep and going back to recalculate a timestep. One reason for not being able to find a solution is close coupling within the circuit.

Methods for determining a new timestep interval [3,4,7,11] use the difference between the actual value of $x_i(t+h)$ and $x_g(t+h)$ to determine whether the node has converged to a solution, and what the best timestep would be for the next timestep. In fact, the maximum value for this difference for all the nodes evaluated during this timestep interval is the number used. These methods use an estimation of the linear truncation error [4,7,11] to determine the new timestep value. The linear truncation error is the difference between the real behaviour of a circuit and the calculated behaviour at

any point in the simulation. The linear truncation error is estimated as the difference between the calculated values, $x_i(t+h)$ and the guessed values, $x_{g,i}(t+h)$.

2.7.3 Algorithms

This section will elaborate on three methods for creating and solving the circuit node equations seen in the last section. The first is one of the direct methods of solution, called Modified Nodal Analysis [4,17]. The second is an iterative method called Iterated Timing Analysis [2,3,8] and the third is also an iterative method called Waveform Relaxation [3,8,18].

2.7.3.1 Modified Nodal Analysis

Modified Nodal Analysis is a special case of Nodal Analysis. Nodal Analysis is a method for generating and solving for the signals in a circuit using matrix solving techniques [19]. The matrix is generated as seen earlier, and can be manipulated and solved using one of the direct methods such as LU decomposition [16]. Other methods are described in the literature [20,21,22]. Modified Nodal Analysis adds the capability of using independent voltage sources and outputting currents to Nodal Analysis [4]. Modified Nodal Analysis solves for the signal value of all circuit nodes at each timestep in the simulation. Even relaxed circuit nodes are evaluated at each timestep. No processing savings are realized when circuit nodes are relaxed.

2.7.3.2 Relaxation Methods

The second and third methods involve de-coupling the system of equations and solving each equation separately. This is done by fixing all variables but one at their current value or best guess, and solving for this one variable. Instead of solving one large system with M variables, the problem becomes one of solving M equations with one variable. All equations are iterated until the differences between each iterated solution fall within specified tolerances (equation (15)).

Relaxation methods decouple the circuit node equations. In an analysis of most circuits it is realized that the rate of change of the signal of each node varies across the circuit. If it could be determined that a node's value will not change significantly at a certain timestep, then with the decoupled equations it would be possible to skip calculating a new value. Such a node is said to be relaxed, and this method of decoupling the equations and not calculating relaxed nodes names this type of method.

2.7.3.2.1 Iterated Timing Analysis

A particular form of the relaxation method is called Iterated Timing Analysis [2,3,8]. This method involves determining, at a timestep, the original value on each node in the circuit based on guessed values on all the other nodes. As a value is calculated for a node, that value becomes the new guess for the node. The method does this one Newton-Raphson iteration on a node at a particular timestep, and then

reschedules that node to be reevaluated at that same timestep. The method then goes on to do one Newton-Raphson iteration on the next node scheduled to be looked at during the current timestep, rescheduling it at the timestep, then iterating around all the nodes to be evaluated at the timestep until each node has converged at the timestep.

Earlier it was stated that the circuit node equations in matrix form could be determined by inspection. After the equations are decoupled, even this process is unneeded. With the decoupled equations, it is necessary to compute the sum of currents entering a node based on the voltage value across the elements connected to that node and to compute the conductance load on the node. This requires element model equations for each element which will give values for the current and conductance as a function of voltage and the element's model's parameters.

2.7.3.2.2 Waveform Relaxation

Another form of the relaxation method is called Waveform Relaxation [3,8,18]. This particular method involves determining the signal on each node from the start to the end of a simulation, based on guessed values of the signals of all the other nodes. This process is repeated using the newly calculated values as the guess values for the other nodes until the solutions converge.

The circuit node equations in matrix form are used, but since all signals except the signal of the node in question are fixed, the solution of the circuit equations become a problem of solving M separate differential equations, each with a single variable. This

can be done using standard numerical techniques. Note that since each node's waveform is calculated separately from each of the others, it is not necessary to synchronize the timesteps of the solution. The best timestep for each node in the circuit can be selected.

2.7.4 Element Models

With each simulation algorithm, the element model equations have to supply the value of the current passing into a circuit node and add a contribution to the conductance on the node. The conductance on a node is determined to be the sum of the input conductances across each of the elements connected to the node.

For the capacitor of Figure 2-1, the current flowing into node b is determined by the equation:

$$i = C \frac{d}{dt}(V_e - V_b) \quad (17)$$

and the conductance supplied to the node is the magnitude of the inductance of a capacitor:

$$G = | \omega C | \quad (18)$$

For small h , then equation (17) becomes equivalent to equation (1), and equation (18) becomes:

$$G = \frac{C}{h} \quad (19)$$

In a similar manner, the conductance seen at each node of a MOS transistor are calculated from the current-voltage equations of the model (based on the SPICE Level I [9] or Shichman-Hodges [23] model) using the following:

At the drain:

$$G_D = \frac{\partial I_{DS}}{\partial V_{DS}} \quad (20)$$

at the source:

$$G_S = \frac{\partial I_{DS}}{\partial V_{GS}} \quad (21)$$

at the gate:

$$G_G = 0 \quad (22)$$

and at the substrate:

$$G_{BS} = \frac{\partial I_{DS}}{\partial V_{BS}} \quad (23)$$

The derived equations are shown at Appendix E.

2.8 Integrated Circuit Layout Characterization

The lowest level of modelling is modelling the functioning of the actual chip layout. This level of detail is not usually of concern to the circuit designer, except where element model parameters are concerned, but is important in the characterizing and proving of the final circuit implementation on a chip. Usually the physical parameters of an element are derived from testing the product of a physical process, and input into models of elements in one of the previous levels of simulation.

2.9 Mixed-Mode Simulators

Level	Relative Cost	Capability and Accuracy
Behavioural	1	Algorithmic verification, some timing information
RTL	10	Functional Verification, some timing information
Gate	100	Functional verification, first-order timing information
Switch	1 000	Functional verification, first-order timing information
Timing	10 000	Detailed waveform information with variable accuracy
Electrical	1 000 000	Most accurate form of simulation

Table 1
Relative Simulator Costs [8,11]

Table 1 shows the relative cost of the progressively more detailed levels of simulation [8,11]. These costs reflect the expense in performing detailed simulations. In effect, electrical simulation is six orders of magnitude more costly than behavioural simulation. Thus it would be wisest to simulate all circuits at a behavioural level. However, behavioural simulators will not always tell the designer the detailed timing information required. The designer could simulate the whole circuit at a lower level, but he or she might not need detailed information for the whole circuit, therefore wasting resources. If the designer could simulate those portions of the circuit where information was required at a more detailed level, and the rest of the circuit at higher levels, the expense of simulation would be reduced. A class of simulators exists which does allow

the designer to simulate circuits at some or all of these levels [1,2,3,8,11,18,24]. These are mixed-mode simulators.

Current research into mixed-mode simulation involves several different tactics. Higher level simulation capabilities are being added to traditional electrical simulators such as SPICE [25,26]. Electrical Simulator algorithms such as enhanced Iterated Timing Analysis [8,14] and variations of Waveform Relaxation [27,28] are also being explored. Parallel computing is being looked at as a way of improving performance [29,30,31], and circuit partitioning and ordering is also being considered [27,28]. The following sections briefly discuss four mixed-mode simulators, their algorithms and other features.

2.9.1 SPICE

SPICE is one of, if not the most common electrical or analog circuit simulators currently in use. SPICE is a general-purpose circuit simulation program for non-linear, DC, non-linear transient, and linear AC analysis. SPICE and its many derivatives implement the Modified Nodal Analysis algorithm [4,8]. Because SPICE uses direct methods to solve the value of a signal on each node in a circuit, a SPICE simulation suffers significantly as the size of a circuit increases [4]. Still, because of its accuracy and features, it is widely trusted and used to verify circuit design [8,9,32].

One commercial variant of SPICE, called PSPICE [25,26], has enhanced the original SPICE by including analog behavioural modelling, event-driven digital

simulation, and extensive model libraries supplying circuit definitions for many thousands of off-the-shelf circuit components [33].

2.9.2 SPLICE1.7

SPLICE is a mixed-mode simulation program developed for the analysis of Metal-Oxide-Silicon Large-Scale Integrated circuits [2,3,8,11,12]. **SPLICE** implements Gate and Electrical levels of simulation, uses the Iterated Timing Analysis algorithm for electrical simulation, and a nine state logic simulator for gate-level simulations [12]. Processing time is reduced over direct methods such as used in **SPICE**, due to the relaxation of nodes and the higher levels of simulation above the electrical level.

A project to add functionality to **SPLICE1.7** in terms of sampled-data models and user-programmable device models [1] has been documented. This project found that adding to an existing simulator required extensive re-engineering of the existing code. In fact **SPLICE1.7** consists of approximately 13 000 lines of **RATFOR** code, with minimal documentation [34]. The **SPLICE1.7** code was found to be poorly designed for modification, with hidden segments of dependent code throughout.

2.9.3 LICE3

iSPLICE3, the latest generation of **SPLICE**, is a mixed-mode simulation program that uses a combination of direct methods and the Iterated Timing Analysis algorithm for

electrical simulation [14,15]. This combination uses the Iterated Timing Analysis algorithm to exploit loosely-coupled portions of the circuit and relaxation, while close-coupled portions of the circuit are solved directly. In addition, iSPLICE3 implements a logic algorithm and a switch-level algorithm called ELogic [13,14,15]. Processing time is reduced over direct methods due to the relaxation technique and abstract levels of simulation above the electrical simulation level.

iSPLICE3 is a completely new implementation of SPLICE1.7, written in C, providing added features, and designed with the future addition of features in mind [8]. One such enhancement is to use results from a previous simulation to speed the simulation of the newly modified circuit. This is called "incremental-in-space" simulation rather than traditional "incremental-in-time" approaches [27,35].

2.9.4 ELDO

ELDO is a circuit simulator which uses an algorithm called the One-Step Relaxation Method [24]. This algorithm is very similar to the Iterated Timing Analysis algorithm, although the element models require fewer calculations of derivatives of the model functions. Like the Iterated Timing Analysis algorithm, the One-Step Relaxation Method requires less memory size than direct methods, meaning larger circuits can be simulated, processing time is proportional to the length of the simulation, and predictive values which reduce iteration time at each calculated timestep can be used.

2.10 Conclusions

It is necessary for circuit designers to verify the functional performance and the correctness of the complex integrated circuits they are designing. They must do the design quickly and accurately, and therefore need tools which can quickly and accurately assist in testing and proving the design. A multi-level simulator will give the designer the advantage of being able to simulate the complex circuit at minimum cost.

When a multi-level simulator is used, the designer can concentrate on simulating the problem areas of a circuit in detail, while using higher level models for more understood areas of the circuit. Table 1 (see page 32) shows the relative cost of detailed electrical simulation. The size and complexity of modern circuits makes electrical simulation costly. A fast, accurate simulation method to support such simulations must be found.

An ideal simulator should have the following characteristics:

1. Accurate representation: the simulator should represent the behaviour of the real circuit such as to be indistinguishable from the real circuit in terms of signal values, timing, and speed.
2. Flexibility. The simulator must permit the circuit designer to simulate what he or she wants to at the level of detail desired and must allow the designer to use abstract methods to represent less critical circuit areas. In addition, the

design must allow for the addition of new simulation algorithms, levels, and device models by third parties.

3. **Speed.** The algorithm(s) used must use some form of relaxation, or some other method of speeding up processing as compared to direct methods.
4. **Accuracy.** The simulator must meet the accuracy needs of the circuit designer.
5. **Low-memory size.** The memory requirements of a simulation must not restrict the usability of the simulator.
6. **Ease of Use.** Although not a primary concern of this thesis, ease of use is a major concern to the circuit designer. The ideal simulator must have a relatively easy interface, and produce results in a useful manner. Concepts of batch processing or interactive processing also fall under this category.

Of these characteristics, the first is the primary target (the perfect simulator should duplicate the behaviour of the real circuit). However, it is not possible to calculate the behaviour of a simulated circuit in the same amount of time in which a real circuit usually behaves. Computer processing speed and the complexity of circuits requires simulating the circuit to some approximation of the real circuit's behaviour. Speed is a concern, considering the size and complexity of modern circuits, yet approximations must be kept to a minimum. Therefore, the ideal simulator would be mixed-mode,

implementing behavioural, gate, timing, and electrical levels of simulation. A relaxation method is a requirement for the backbone of the simulator. Accuracy is important. The simulator must implement an accurate simulation algorithm. The simulator must be easy to use, and ideally it would present useful output as soon as possible, and moreover would present any problems in the circuit design as early as possible. This would suggest some form of interactive output to approach the target, the behaviour of a real circuit which allows interactive probing and tuning.

This discussion reveals that a realistic, sound simulator would be a mixed-mode simulator implementing a relaxation-based simulation algorithm. Work constraints including available time and a one-person project require consideration before the algorithms and modes for a mixed-mode simulator can be chosen. A flexible object-oriented design and programming environment will increase the effectiveness of the work effort. The simulator model of Figure 2-3 (see page 10) will contribute to the flexibility of the simulator design. Algorithms and modes that are easily implementable will be chosen within the work constraints. Iterated Timing Analysis is one such relatively simple, accurate and flexible algorithm that can be readily implemented. Behavioural-level sources, gate-level, and electrical simulation give a minimum flexibility to the simulator. The remainder of this thesis discusses the development and implementation of just such a simulator [36].

Chapter 3 ROOMMS - A Relaxation-based, Object-Oriented, Mixed-Mode Simulator

3.1 Introduction

This chapter describes the implementation of ROOMMS, a Relaxation-based, Object-Oriented, Mixed-Mode Simulator. ROOMMS was designed to explore the concept of mixed-mode simulators, especially in support of the telecommunication circuit design community. In addition, ROOMMS was designed so that it could be enhanced by a simulator developer in order to explore new algorithms and other simulation concepts.

ROOMMS implements the simulator model presented in Section 2.1.2 and Figure 2-3. The intermediate level logic simulation algorithm and models also presented in that chapter are implemented. The electrical level Iterated Timing Analysis algorithm and device models are also implemented in ROOMMS, as shown in this chapter.

As well as presenting theory and background material, Chapter 2 described an ideal simulator. ROOMMS is not that ideal simulator, but rather was used to explore some of the features of such a simulator. ROOMMS is a mixed-mode simulator implementing behavioural-level sources, gate and electrical levels of simulation. It implements the relaxation-based Iterated Timing Analysis algorithm, which is flexible, fast and accurate. ROOMMS does not implement more than a rudimentary user interface and operates in batch mode only.

ROOMMS has been written in the object-oriented programming language - Objective C [37]. The object-oriented concepts of classes, encapsulation, hierarchical organization and polymorphism (see Appendix A) speed up the programming process, and have helped the creation of a complex program that is hierarchically structured, compartmentalized, easily modified, and understandable.

ROOMMS implements the Iterated Timing Analysis [2,3] algorithm, and packages this proven analog algorithm with a proven relaxation-based logic algorithm [2]. Powerful signal generators or abstract sources are also implemented which can drive the electrical and gate level simulation with signals on a behavioural level. Finally, the design of ROOMMS allows the addition of new features to the simulator with relative ease.

3.2 Algorithms

3.2.1 Logic Simulator Algorithms

The proven logic algorithm introduced in Section 2.4.1 implements a three-level signal (high, low and unknown) with driving strengths and gate delays. The simulator model of Figure 2-3 is implemented. The logic node process implements the algorithm of Figure 3-1. When a node is processed, it determines a new state based on all its input elements (fanins) and calculates the delay on the node. This is done by looking at each fanin element in turn. The state of the node is determined by looking at the input nodes

```

start
  for each fanin element of the node do {
    determine output state
    determine the delay associated with this state
    determine whether this output state "wins" the node
  }
  if the winner state is different than the old state on the node {
    schedule the logic node with the winner state and delay
  }
} done

```

Figure 3-1
Logic Node Processing Algorithm

of that element and by using a look-up table, determining the output on the node. The delay through the element is calculated. Then the node determines whether that element's output will drive the node. This is done by using the strength portion of the state. If the element is the strongest one driving the node at any particular time, it will win the node, and the output of that element will become the new value on the node after the delay.

If equal strength elements are driving the node and if the logic value is the same, the output is known, and if the logic values are different, then the node will be given an unknown value.

With a logic algorithm implementing a delay, it is possible for a node's value to be scheduled to change in the future. If another stimulus to a fanin element of the node occurs affecting the current state of the node, there is the danger that a non-valid signal, a glitch, will be propagated through the circuit [2]. The scheduling algorithm of Figure 3-2, implemented in the scheduler process of Figure 2-3, is used to identify glitches and re-schedule the node properly so that the glitch is not propagated. A node is only scheduled once at any particular time during the simulation. When a glitch is

```

begin {
  if lastTime < PresentTime do {
    store new state in node
    schedule node at nextTime
  }
  else if lastTime == PresentTime do { // this means the node is scheduled now
    if nextTime >= lastTime do {
      if node has been processed do { // note that a glitch
        update new state on node // has occurred, and not ignored
        schedule node at nextTime
      }
      else { // node has not been processed
        drop scheduling at lastTime
        replace new state on node
        schedule node at nextTime
      }
    }
  }
  else if lastTime > PresentTime do { // node is already scheduled in future
    if nextTime < lastTime do {
      // reschedule time is earlier than originally scheduled time
      // you've just caught a glitch
      report glitch
      drop scheduling at lastTime
      replace new state on node
      schedule node at nextTime
    }
    else if nextTime == lastTime do {
      report glitch
      replace new state on node
    }
    else if nextTime > lastTime do { // schedule in future
      report glitch
      drop scheduling at lastTime
      replace new state on node
      schedule node at nextTime
    }
  }
} // done

```

where PresentTime is now,
 nextTime is next time node is to be scheduled, and
 lastTime is last time node was scheduled

Figure 3-2
Logic Node Scheduling Algorithm

identified, the node is rescheduled at the new delay and with the new state. The algorithm is not perfect however, and under certain conditions, the glitch will propagate through the circuit.

The propagation of a glitch occurs when a node is evaluated at a timestep and its new value propagated (the glitch), and then its value changes at a later iteration during the same timestep. This glitch is identified, but is propagated since the evaluation already took place. All other glitch conditions are caught and all these other glitches are not propagated. The glitch that is propagated occurs depending on the order in which the node is evaluated at that time. If the node which potentially has a glitch is evaluated (due to the scheduling delay which is going to cause the potential glitch) before its fanin element's input nodes, then the glitch will occur and will be propagated. ROOMMS schedules input sources' output nodes first, which partially insures that these types of glitches will not occur. No further processing to order the nodes to be evaluated by the scheduler just to prevent this type of glitch is undertaken.

3.2.1.1 Logic Simulator Delay Modelling

ROOMMS implements delays determined by specified parameters of a gate model and from the static loading on any specific node [2,12]. The user defines the basic gate delay of a gate, the input and output capacitive load, and the output driving capability of

the gate. These last are called driving factors. The total delay of a signal on a node is then given by:

$$risedelay = tr + trc \times (totaloutputnodecapacitance) \quad (24)$$

or

$$falldelay = tf + tfc \times (totaloutputnodecapacitance) \quad (25)$$

where

risedelay, *falldelay* are the delays before the signal on the node changes,

tr, *tf* are the rise and fall delays of the gate

trc, *tfc* are rising and falling driving strengths of the gate

3.2.2 Analog Simulator Algorithm

ROOMMS implements the Iterated Timing Analysis (ITA) algorithm described in Section 2.7.3.2.1. This algorithm is implemented in the analog node process of Figure 2-3. The ITA algorithm is shown in Figure 3-3. When a node is processed, a new voltage value is calculated (one Newton-Raphson iteration), and if the voltage change between the last iteration at this timestep and the current one is significant, then the node has not converged and is scheduled for re-evaluation at the current timestep using this last value calculated as the new guess voltage. If the node has converged, then it is determined whether this new voltage is significantly different from the voltage on the

```

begin {
  obtain node
  if first time processed at PresentTime {
    use last two points to make guess on new voltage on node
  }
  TotalConductance = 0
  TotalCurrent = 0
  TotalCapacitance = capacitance on node
  for each fanin element of node do {
    compute equivalent conductance
    compute total current flowing into the node
    compute total capacitance on node
  }
  calculate deltaV
  currentVoltage = previousGuess + deltaV
  voltageChange = currentVoltage - previousVoltage
  if deltaV < tolerance do { // node has converged
    if not converged at this time before do {
      if voltageChange is significant do {
        schedule current event at PresentTime + 1
        schedule node at now
      }
      else { // node change is not significant - do nothing
      }
    }
    else { // have converged before at time now
      // do nothing, to break feedback loops
    }
  }
  else { // node has not converged, so keep processing
    schedule current event now
    schedule node now
  }
} // finished this iteration

```

Figure 3-3
Iterated Timing Analysis Algorithm

node at the last timestep when the node was processed. If the change is significant, then the node is scheduled to be evaluated at the next timestep, and if there is no significant change, the node is allowed to relax.

3.3 Device Models

The Elements process of Figure 2-3 is responsible for determining a new contribution of that element's output to the node when the node is being processed. ROOMMS implements three main types of elements: analog elements such as MOS transistors for the electrical or analog simulator; logic gates for the logic simulator; source elements for both types of simulation; and one special element, the grounded capacitor.

3.3.1 Analog Device Models

As described in Section 2.7.4, the analog device models implement equations which supply a value of current passing into a circuit node and add a contribution to the conductance load on that node. Each of the equations used to implement the device models of ROOMMS are contained in Appendix E.

3.3.2 Logic Device Models

Logic device models supply a new logic value and strength to the logic node process. In addition, each element has a characteristic delay. ROOMMS determines the output of simple gates such as AND and OR gates, by using look-up tables. ROOMMS has implemented more complex gates such as an edge-triggered, clocked, set-reset D flip-flop with enable with direct processing savings to determine the output states. Sensing the edge-triggering requires significant processing that overshadows any processing savings that would come from look-up tables.

3.3.3 Source Elements

ROOMMS implements powerful source elements which are used to implement input signals into the circuit being simulated, and to model more complex elements of a circuit. A source is a special element because the output of the source element forces a signal onto a node. This requires the assumption that the source's output signal is driven more strongly than any feedback signal in the circuit under test. It is also assumed that only one source drives a node.

ROOMMS implements logic and voltage rails, timed sources based on piecewise-linear pulse signals for logic and analog simulators and sinusoidal and polynomial driven sources for analog simulation. In addition, ROOMMS implements a set of controlled sources. These include logic-controlled voltage sources,

voltage-controlled logic sources, voltage-controlled voltage sources, and a logic-clocked, voltage-controlled voltage source.

3.3.4 Grounded Capacitor

ROOMMS implements a grounded capacitor in a special manner due to the way the node algorithms are implemented. Each node has the property of being a capacitive load on a fanin element. Therefore a node has a capacitive load value equivalent to the sum of the loads presented to the node by the elements attached to the node and any grounded capacitor on the node. This is true for both analog and logic modes of the simulator. Therefore, the capacitance of a grounded capacitor is added to a node during the setup phase of the simulator, and the grounded capacitor element itself is not used during the simulation phase of the simulator.

The logic simulator delay algorithm uses the capacitive value to calculate a delay, as seen in Section 3.2.1.1.

The analog simulator algorithm linearizes the load this capacitive value applies to the node by dividing the capacitive value by the small timestep, and adding the result to the total conductance on the node.

3.4 Implementation of ROOMMS

3.4.1 Program Organization

ROOMMS implements the simulator model of Figure 2-3 (see page 10) in the object-oriented programming language Objective C. The processes of this model translate directly into Objective C's programming constructs.

Objective C programs are organized in term of classes (see Appendix A), which are the primary organization of an object-oriented programming language. A hierarchy of classes describes the program's organization. ROOMMS is implemented in this manner and its class structure is introduced in a simplified form in Figure 3-4 (detailed class descriptions are given in Appendix B).

Figure 3-4
Simplified Class Hierarchy

Figure 2-3, the simulator model, shows the main processes of a simulator. Figure 3-4 shows the core of ROOMMS, and the implementation of the processes of Figure 2-3. The Simulation class contains the controlling processes of the simulator.

This class sets up the simulation's data structures, and then controls the simulation by implementing the Timestepper and Handle Timestep processes of Figure 2-3.

The Scheduler class defines the scheduler process of Figure 2-3. Contained in the definition of this class is the data structure that represents the scheduler queue or event wheel.

The Elements class and its subclasses implement the element process of Figure 2-3. Figure 3-4 shows a sub-hierarchy of classes under the Elements class. Elements contains the common data structures and methods of analog and logic elements. LogicElements inherits those data structures and methods from Elements and defines the special behaviour of logic elements in its methods. AnalogElements inherits the same data structures and methods from Elements and defines the special behaviour of analog elements. As was stated in Section 3.3.4, the grounded capacitor element is a special case. The Gcapr class shares the common data structure and methods of all the other elements, but since the special use of grounded capacitors means that these elements are not processed the same way as other elements, grounded capacitors do not need the additional procedures of the AnalogElements and LogicElements classes.

The Node process of Figure 2-3 is implemented in the Nodes class of Figure 3-4. Nodes contains the common data structures and methods of the node types. LogicNodes inherits those common data structures and methods and defines the logic simulator specific procedures. The logic simulation algorithm described in Section 3.2.1 is implemented in the LogicNodes class. AnalogNodes inherits those common data structures and methods and defines the analog simulator specific procedures. The analog

simulation algorithm. Iterated Timing Analysis, discussed in Section 3.2.2 is implemented in the AnalogNodes class.

The Source class and its subclasses implement the processing of the source elements.

Figure 3-5
Elements Class Hierarchy

Figure 3-5 shows the full sub-hierarchy of subclasses of the Elements class, including each of the element types currently implemented in ROOMMS.

The logic element classes: Or, And; and Xor, contain the look-up tables that give the output of these elements. The inverse elements of these: Nor, Nand; and Xnor, use the same look-up tables, but invert the output signals. The Inverter element, and the Buffer and Delay elements, are single-input, single-output, and the processing detail is even simpler. The Flip-flop elements, implementing D and JK FlipFlops, have more complex algorithms due to their edge-triggering behaviour.

The analog element classes: Tres (resistor) and Fcapr (floating capacitor); and the MOSFETS: Nmos and Pmos, are implemented in their respective classes. Section 3.3.1 explained that the analog elements calculate a current and conductance increment when a node is evaluated at a timestep. As stated in Section 3.3.1, the MOSFET model implemented in the Mosfet class is the Shichman-Hodges, SPICE Level 1 model [9,23]. An arbitrary current source is implemented within the Isourceelement class. This current source could be implemented with any of the characteristics of the other sources, but is currently a constant current source.

Figure 3-6
Sources Class Hierarchy

Figure 3-6 shows the sub-hierarchy of subclasses of the Sources class. The source elements include controlled sources: Vcls (voltage-controlled, logic-source), Lcls (logic-controlled, voltage-source); Vcls (voltage-controlled, voltage-source); and Lcvcls (logic-controlled, voltage-controlled, voltage-source), timed sources: Lsrc (logic source); Tsrc (an analog source); Sinusoid (an analog source); and General (a polynomial-driven analog source), and constant sources: Lrail (constant logic) and Vrail (constant analog).

Figure 3-7
Nodes Class Hierarchy

Figure 3-7 shows the sub-hierarchy of subclasses of the Nodes class. As was seen before, there are LogicNodes and AnalogNodes classes. Additional classes are shown which implement the special behaviour of nodes which are output nodes of sources. These nodes have a value which is driven by one of the subclasses of the class Source. In addition, they have the special characteristics of being scheduled at the head of the Scheduler's queue. Scheduling these nodes first means that the sources' output is handled before the nodes in the subcircuit driven by the source are handled, reducing the number of iterations in the analog simulator, and reducing the possibility of uncatchable glitches in the logic simulation.

Figure 3-8 shows the full class hierarchy of ROOMMS which includes all the classes just seen, and a few more. The class Object represents the generic built-in behaviour of all classes of the Objective C language. SpecialDependencyObject defines utilities which realize the connections between elements and nodes.

Figure 3-8
ROOMMS Class Hierarchy

There are a few classes associated with the class Simulation, seen in Figure 3-4. SimulationUtilities defines some utility methods. CircuitInput handles the input of a circuit description of the form shown in Figure 2-2 and Appendix D. This particular class was implemented as a separate project [38] in the programming language "C", combined with a few Objective C calls.

The two classes at the top right of Figure 3-8 implement the LogicState, which contains the data structure that holds the logic value and logic strength present on a node, and Increments, which describes the data structure which holds the conductance, current and capacitance on an analog node while it is undergoing evaluation at any particular timestep.

Four classes shown on Figure 3-8, Diode, Bjt, Npn, and Pnp, represent analog element classes currently not implemented in ROOMMS.

3.4.2 Implementation History

The work on ROOMMS was preceded by a programming exercise to test the feasibility of using an object-oriented programming language. The initial experiment was a no-name implementation of the logic algorithm in the SMALLTALK-derived language "METHODS" [39]. This experiment was successful, developing a prototype logic simulator in less than two months work. METHODS is an interpreted language. Unfortunately, an interpreted language must translate and execute each line of a program

as it comes to it. Therefore, METHODS pays the price of a flexible programming environment with the poor performance inherent in interpreted code.

ROOMMS itself was developed in Objective C to reap the benefits of the speed of a compiled language over the METHODS implementation. While Objective C is also a SMALLTALK derivative, significant translation was required to port the METHODS code over to Objective C. Also, the METHODS code consisted of a fancy user interface with menus and multiple information windows on a PC environment with the logic simulation mode implementation. The development platform for what was to become ROOMMS was chosen to be a SUN, and the decision was made not to attempt to duplicate the user interface of the METHODS version, but rather to concentrate on the functionality of the simulator. Revising the logic algorithm implementation, adding the analog algorithm, and enhancing the overall simulator took a further 12 months of effort.

The initial class hierarchy for the ROOMMS simulator is shown in the top left hand corner of Figure 3-9. This can be compared to the present class hierarchy structure in the bottom left hand corner. This figure shows two intermediate stages between the first class hierarchy and the current one to show stages in the maturity of ROOMMS. Initially, the simulator had many functions implemented within the Simulation class and its superclasses. At this point in the development the simulation model of Figure 2-3 (see page 10) did not exist.

The first stage in the development of ROOMMS was to think. One result was the simulation model. With this model, it became clear that the simulator could be organized more logically. The result is the second class hierarchy which shows separate Scheduler,

Figure 3-9
ROOMMS Class Hierarchy Development

Nodes, Elements and Simulation classes. Analog simulation has been added, which resulted in separate subclasses of Elements and Nodes. Some classes were implemented in a more independent fashion, and moved up in the class hierarchy. Also at this time, a separate project was started to develop a input interface to the simulator [38]. The initial intent of this project was to interface PDOMMS with an existing graphics package which contained a schematic capture feature, although that part of the separate project was too ambitious. It was planned to use an intermediate file format which would be useful as an interim input form. The current CircuitInput class, as developed under that separate project, parses that intermediate file format and loads the information into the simulator data structures.

Between the second and third class hierarchies, the concept of sources was rethought. If nodes driven by sources were to be scheduled before all other nodes, these nodes which are forced to a value would be evaluated first, and therefore all the following node evaluations would take place with the proper source signals. This reduces the wasted computations if a non-source node was evaluated before a source node which had an effect felt by that non-source node. The result was separate source input and output nodes which were scheduled specially by the Scheduler.

Finally, the source classes were seen to be sufficiently independent of the implementation of the Nodes class, and the source classes were raised in the class hierarchy. It was also determined that there was no need to distinguish between source input nodes and source output nodes. These classes were merged. Additional device

models were added, and the result is the class hierarchy of Figure 3-8 and shown at the bottom left of Figure 3-9.

The current implementation of ROOMMS is capable of simulating complex circuits as will be shown in the following Chapter. However the performance of ROOMMS is not good. In comparison to the existing mixed-mode simulator SPLICE1.7 [12] which implements the same algorithms, ROOMMS is ten times slower during analog simulation. The main use of time in the ROOMMS simulator's analog simulation is the iteration loop in the Node process which determines the signal value on a node. This iteration loop is included within the definition of the AnalogNodes class, and makes message calls to the analog element classes and the Increments class. Each of these messages result in a context switch (i.e. at a primitive level in the computer, registers are pushed on and off a stack). While an object-oriented program consists of message calls exclusively, Objective C is based on the "C" programming language, and therefore it is possible to reduce the number of message calls within the tight iteration loop by using C code. ROOMMS was developed using the "object-oriented" mindset, and unnecessary overhead exists in the main iteration loop which could be trimmed, but which was not due to time constraints of this project.

The first task in increasing the performance of ROOMMS would be to attack this problem of too many message calls during the tight iteration loop of the analog simulation. The class Increments should be replaced with a "C" record structure. This would involve modifying the AnalogNodes class, the AnalogElements class and each of

AnalogElements' subclasses. This change should significantly increase the performance of ROOMMS, while not significantly affecting the maintainability of the code.

3.5 Adding Functionality to ROOMMS

As an example of the ease of adding functionality to ROOMMS, the addition of the timing simulation or quasi-analog capability, ELogic [13,14,15], can be considered. Adding this capability requires implementing a new version of the Node and Element processes of the Figure 2-3 simulator model. This would be done by creating a small number of new classes which would define the new data structures and algorithm. These new classes would include ELogicNodes, ELogicElements, and a device model class for each device.

ELogicNodes would implement the Node process. This would be very similar to the AnalogNodes class in its implementation, but would differ in the way a signal value is determined and how an event is scheduled. This close similarity between AnalogNodes and the new ELogicNodes means that most of the behaviour of AnalogNodes could be used by ELogicNodes. Therefore, ELogicNodes would be implemented as a subclass of AnalogNodes. ELogicNodes would redefine only the method which determines the new voltage on a node and would inherit all other behaviour and all data structure from AnalogNodes (see Appendix B).

ELogicElements and its subclasses would implement the Element process. Because of the discrete nature of the response of the ELogic device models, table look-up

models would be used, reducing the computations required during the simulation, at the expense of some pre-simulation overhead and data storage. The data structures and most of the behaviour of the ELogicElements class would be inherited from the Elements superclass. Only the handling and storage of the look-up table would need to be defined in ELogicElements. The subclasses of ELogicElements (that is, the individual ELogic device models) would contain the behaviour definition required of each device to load the device's look-up table. Any of the analog device models used in the analog simulation, as well as more abstract models, could be defined.

Very little of the rest of the simulator would have to be touched in order add the ELogic implementation. Most notable would be an initialization method in the class Simulation called "newElementOfClass", and possibly the implicit conversion routines in AnalogNodes and LogicNodes. The main program of the simulator would have to be updated so that the new classes are recognized by the rest of the simulator.

Only two main classes and the device models need to be developed, and minor changes need to be made to very few of the remaining class definitions.

It is estimated that the addition of the ELogic capability requires the development of less than 400 lines of code (plus the device models).

3.6 Summary

This chapter describes the implementation of ROOMMS. It shows the major algorithms of ROOMMS, and the hierarchical program design and implementation.

ROOMMS was designed in order to support the telecommunications circuit designer community, and to allow enhancements at a future date. This chapter ends by briefly discussing a possible enhancement of ROOMMS showing the relatively small effort required for the enhancement. The following chapter will show the ROOMMS simulation results of an in-service telecommunications circuit, and describe the testing effort followed in order to verify the functionality of ROOMMS.

Chapter 4 Testing Results from ROOMMS

Testing and verification of the functioning of the ROOMMS simulator involved extensive use of both simple and complex circuits. Testing involved comprehensive trials of isolated portions of the simulator in order to see that data structures were created and contained the correct information, and the testing of each implemented gate in simple circuits. Once the functionality of the code was verified, more complex circuits were simulated and the results compared to hand-calculations and SPICE and SPLICE results. This chapter shows three circuits and their results, a boot-strapped inverter example, a mixed-mode element of a transmit filter portion of an in-service telecommunications circuit [6,40], and the analog signal reconstruction part of that filter.

A boot-strapped inverter [12] is presented in Figure 4-1. This shows the capability of ROOMMS to handle a circuit with a large floating capacitor, which results in very close coupling of two nodes in the circuit. Figure 4-1 shows this circuit and the ROOMMS source deck, while Figure 4-2 shows the ROOMMS output compared to HSPICE and SPLICE1.7 results.

The next circuit is a mixed-mode circuit portion of an in-service transmit filter [6,40] which samples a biphas input (represented by the logic sources q_i and q_{n-i}) and which drives one coefficient (r_1 and b_1) of the analog signal reconstruction section shown in Figure 4-6. The mixed-mode example is shown in Figure 4-3. The output is shown in Figure 4-4.

Figure 4-1
Boot-strapped Inverter

Figure 4-2
Boot-strapped Inverter Output

Figure 4-3
Mixed-Mode Circuit

Figure 4-4
Mixed-Mode Circuit Output

Figure 4-5
Folded Cascode Amplifier of Filter

The analog signal reconstruction portion of the transmit filter [6,40] is shown in Figures 4-5 and 4-6. Figure 4-5 shows the folded cascode amplifier, and Figure 4-6 the remainder of the circuit (Appendix D contains the input source deck for this circuit). The mixed-mode portions of the circuit (each of which are similar to Figure 4-3) feed the signals r1, r2, r3, ... r8 and b1, b2, b3, ... b8 into the folded cascode opamp low pass filter. The output of this analog part of the circuit, which includes 35 MOS transistors,

Figure 4-6
Switched-Capacitor Pulse Shaping Filter

11 floating capacitors and an RC load, are shown in Figure 4-7, compared to HSPICE. Note that timing sources were used to represent the behaviour, as verified by the simulation of the mixed-mode part of the filter. In Figure 4-7, the virtual ground (2), opamp output (3) and filter output (29) are shown for ROOMMS (R) and HSPICE (S). The output waveforms of Figures 4-2 and 4-7 clearly show the agreement between ROOMMS output and HSPICE in terms of the general behaviour of the output after initial settling has occurred.

Chapter 5 Conclusions

5.1 Summary

The objective of this thesis was to explore the concept of simulators and to determine what an ideal simulator should be in order to meet the needs of the telecommunication and other circuit design communities. Among various individual levels of simulation, the concept of mixed-mode simulation was shown to be the best solution for simulating modern complex circuits. The theory behind mixed-mode simulators was presented. Included in this description was the development of a simulator model, followed by a review of current practice. An ideal simulator was presented and discussed and the development of a mixed-mode simulator capable of supporting the design of telecommunications circuits was then described. In addition, it was shown how the development environment and logical design eases the work involved in adding functionality to the simulator. The testing and verification of that simulator, ROOMMS, was presented, including the simulation of an actual telecommunications circuit. From these simulations, ROOMMS was shown to have the functionality required to support the needs of the telecommunications and other circuit design communities.

5.2 Achievements and Contributions

The ROOMMS simulator, a Relaxation-based, Object-Oriented, Mixed-Mode Simulator, was developed with a sound simulator architecture which can be used to explore the concepts of mixed-mode simulation. As a working, practical, mixed-mode simulator capable of behavioural, gate and electrical simulation of telecommunications and other circuits in the time domain, this basic mixed-mode simulator has been designed and implemented in a manner which will allow it to be used as a framework for further implementation of simulation levels.

While ROOMMS is not the ideal simulator presented in Section 2.10, it implements those features of the ideal simulator that could be implemented within the scope of this work. ROOMMS is flexible, implements a relatively fast algorithm, is accurate and does not use excessive memory. As such this simulator fits within the functionality range of modern mixed-mode simulators. While the user-interface is primitive, this was not of concern during this work.

The use of one of the latest object-oriented software development environments and the proposed simulator model was shown to ease the process of simulator development. The amount of work accomplished was directly related to the ease of development in this environment. Object-oriented design and development resulted in a logical design that is flexible with respect to being able to add new algorithms and device models, and yet quite comprehensive considering the available work effort.

The flexibility of the design of ROOMMS was shown with an intellectual exercise involving the addition of another simulation level, the timing simulation algorithm ELogic. This analysis showed that minimal coding effort is required to implement ELogic within the framework of ROOMMS.

Finally, a purpose of this simulator development was to explore the use of this type of development environment with respect to developing a simulator, simulation algorithms and device models. The logical design required by this environment and the ease with which enhancements to the simulator can be made show that there are significant benefits to be gained from using an object-oriented design and development approach to this type of work.

5.3 Suggestions for Further Work

The simulator presented in this thesis consists of the core of a practical simulator. Its sound, flexible architecture will allow additional powerful features to be added, permitting further areas of simulation theory to be explored in detail. The existence of a capable simulator framework will permit this to take place.

The work accomplished as part of this thesis met its goal of developing a working simulator capable of simulating telecommunications circuits. However, time did not permit a thorough exploration of several areas, such as the timing simulation mode ELogic, and additional analog algorithms such as Waveform Relaxation.

It was chosen to not explore input and output application areas in this thesis. A workable first cut was implemented, but significant work remains in this area.

An intriguing concept to explore would be the addition of an interface to externally defined user programs, permitting the investigation of parallel processing through closely synchronized simulation programs.

References and Bibliography

- [1] M.A. Copeland, G.P. Bell, and T.A. Kwasniewski: "A Mixed-Mode Sampled Data Simulation Program", IEEE Journal of Solid-State Circuits, Vol. SC-22, No. 6, Dec 1987, pp. 1098-1105.
- [2] R.A. Saleh: "Iterated Timing Analysis and SPLICE1", Memorandum No. UCB/ERL M84/2, University of California, Berkeley, 1984.
- [3] R.A. Saleh: "Nonlinear Relaxation Algorithms for Circuit Simulation", Memorandum No. UCB/ERL M87/21, University of California, Berkeley, 1987.
- [4] P. Antognetti, D.O. Pederson, and H. De Man, "Computer Design Aids for VLSI Circuits", Martinus Nijhoff Publishers, Boston, 1986.
- [5] A.R. Newton, "The Simulation of Large-Scale Integrated Circuits", Memorandum No. UCB/ERL M78/52, University of California, Berkeley, July 1978.
- [6] R.P. Colbeck, and P.B. Gillingham: "A 160-kb/s Digital Subscriber Loop Transceiver with Memory Compensation Echo Canceller", IEEE Trans. on Circuits and Systems, Vol. CAS-33, No. 2, Feb 1986, pp. 175-182.
- [7] J. Vlach, and K. Singhal, "Computer Methods for Circuit Analysis and Design", Van Nostrand Reinhold Company, New York, 1983.
- [8] R.A. Saleh, and A.R. Newton, "Mixed-Mode Simulation", Kluwer Academic Publishers, Boston, 1990.
- [9] P. Antognetti, and G. Massobrio, "Semiconductor Device Modelling In SPICE", McGraw-Hill Book Company", Toronto, 1988.
- [10] C.M. Class, and D.K. Frederick, "Modelling and Analysis of Dynamic Systems", Houghton Mifflin Company, Boston, 1978.
- [11] J.E. Kleckner, "Advanced Mixed-Mode Simulation Techniques", Memorandum No. UCB/ERL M84/48, University of California, Berkeley, June 84.
- [12] R.A. Saleh, J.E. Kleckner, and A.R. Newton: "SPLICE Version 1.7 User's Guide", Department of Electrical Engineering and Computer Services, University of California, Berkeley, 1983.
- [13] Y.H. Kim, S.H. Hwang, and A.R. Newton: "Electrical-Logic Simulation and Its Applications", IEEE Trans. on Computer-Aided Design, Vol. 8, No. 1, Jan 1989, pp. 8-22.

- [14] E.L. Acuna, J.P. Dervenis, A.J. Pagones, R.A. Saleh: "ISPLICE3: A New Simulator for Mixed-Analog/Digital Circuits", Proc. of IEEE 1989 Custom Integrated Circuits Conference, pp. 13.1.1-13.1.4.
- [15] R.A. Saleh, "ISPLICE Version 3.0 User's Guide", Department of Electrical and Computer Engineering Coordinated Science Laboratory, University of Illinois, 1989.
- [16] W.H. Press, B.F. Flannery, S.A. Teukolsky, and W.T. Vetterling, "Numerical Recipes in C - The Art of Scientific Computing", Cambridge University Press, New York, 1988.
- [17] C-W. Ho, A.E. Ruehli, and P.A. Brennan, "The Modified Nodal Approach to Network Analysis", IEEE Transactions on Circuits and Systems, Vol. CAS-22, No. 6, June 1975, pp. 504-509.
- [18] E. Lelarsmee, A.E. Ruehli, and A.L. Sangiovanni-Vincentelli, "The Waveform Relaxation Method for Time-Domain Analysis of Large Scale Integrated Circuits", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-1, No. 3, July 1982, pp. 166-182.
- [19] L. Nagel, R. Rohrer, "Computer Analysis of Non-linear Circuits, Excluding Radiation (CANCER)", IEEE Journal of Solid-State Circuits, Vol. SC-6, No. 4, August 1971, pp. 166-182.
- [20] K. Yamamura, K. Horiuchi, "A Global and Quadratically Convergent Algorithm for Solving Nonlinear Resistive Networks", IEEE Transactions on Computer-Aided Design, Vol. 9, No. 5, May 1990, pp. 487-499.
- [21] F.H. Branin, Jr., G.R. Hogsett, R.L. Lunde, L.E. Kugel, "ECAP II - A New Electronic Circuit Analysis Program", IEEE Journal of Solid-State Circuits, Vol. SC-6, No 4, August 1971, pp. 146-166.
- [22] W.T. Weeks, A.J. Jimenez, G.W. Mahoney, D. Mehta, H. Qassemzadeh, T.R. Scott, "Algorithms for ASTAP - A Network-Analysis Program", IEEE Transactions on Circuit Theory, Vol. CT-20, No. 6, November 1973, pp. 628-634.
- [23] H. Shichman, and D.A. Hodges, "Modelling and Simulation of Insulated-Gate Field-Effect Transistor Switching Circuits", IEEE Journal of Solid-State Circuits, Vol. SC-3, No. 3, Sept. 1968, pp. 285-289.
- [24] B. Hennion, and P. Senn, "ELDO: A New Third Generation Circuit Simulator Using the One-Step Relaxation Method", Proceedings of ISCAS 85, 1985, pp. 1065-1067.

- [25] The Design Center Newsletter, Microsim Corporation, April 1992.
- [26] The Design Center Newsletter, Microsim Corporation, July 1992.
- [27] Y-C. Ju, R.A. Saleh, "Incremental Circuit Simulation Using Waveform Relaxation", 29th ACM/IEEE Design Automation Conference, 1992, pp. 8-11.
- [28] Y-H. Jun, C-W. Lee, K-J. Lee, S-B. Park, "Timing Simulator by Waveform Relaxation Considering the Feedback Effect", IEE Proceedings, Vol. 136, Pt. G, No. 1, February 1989, pp. 38-42.
- [29] G.G. Hung, Y.C. Wen, K. Gallivan, R. Saleh, "Parallel Circuit Simulation Using Hierarchical Relaxation", 27th ACM/IEEE Design Automation Conference, 1990, pp. 394-399.
- [30] D. Dumlugöl, P. Odent, J.P. Cockx, H. De Man, "Switch Electrical Segmented Waveform Relaxation for Digital MOS VLSI and Its Acceleration on Parallel Computers", IEEE Transactions on Computer-Aided Design, Vol CAD-6, No. 6, November 1987, pp. 992-1005.
- [31] E.Z. Xia, R.A. Saleh, "Parallel Waveform Newton Algorithms for Circuit Simulation", IEEE Transactions on Computer-Aided Design, Vol. 11, No. 4, April 1992, pp. 432-442.
- [32] D.B. Herbert, "Variety is the Life of SPICE", Electronic Design, June 25, 1992, pp. 53-66.
- [33] Microsim Corporation, "The Design Center", Product Brochure, Microsim Corporation, Irvine, California, Jan 92.
- [34] J.A. Watts, "Development of SPLICE as a Design Tool for Analog Circuits, and Implementation of a Frequency Post-Processor for SPLICE", 4th year Project Report, Department of Electronics, Carleton University, 1985.
- [35] Y-C. Ju, F.L. Yang, R.A. Saleh, "Mixed-Mode Incremental Simulation and Concurrent Fault Simulation", Proceedings of the ICCAD, Nov 90.
- [36] J.A. Watts, and T.A. Kwasniewski, "ROOMMS, A Relaxation based, Object-Oriented, Mixed-Mode Simulator", Proceedings of the IEEE 1990 Custom Integrated Circuits Conference, 1990, pp. 5.3.1-5.3.4.
- [37] B.J. Cox: "Object-Oriented Programming An Evolutionary Approach", Addison-Wesley Publishing Company, Don Mills, Ontario, 1986.

- [38] R. Ivan, Department of Electronics, Carleton University, work undertaken during 4th year project, Department of Electronics, Carleton University, 1989.
- [39] Digitalk Inc., METHODS V1.0, 1984.
- [40] R.P. Colbeck, Mitel Corporation: private correspondence, 1989.
- [41] A. Goldberg, and D. Robson: "Smalltalk-80 The Language and its Implementation", Addison-Wesley Publishing Company, Don Mills, Ontario, 1983.
- [42] A. Vladimirescu, K. Zhang, A.R. Newton, D.O. Pederson, "SPICE Version 2G 4 User's Guide", Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1981.
- [43] Canadian Microelectronics Corporation, "Guide to the Integrated Circuit Implementation Services of the Canadian Microelectronics Corporation", GIGCIS Version 2:0, January 1986.

Appendix A Software Development Environment

Background

A.1 Objective C and SMALLTALK

ROOMMS has been written in the object-oriented programming language - Objective C, an implementation of the programming environment SMALLTALK [37,41]. Unlike SMALLTALK, Objective C is not an environment: it rather borrows many of the useful features of SMALLTALK and implements them as a superset of the conventional programming language 'C'. These features are: objects, classes, message passing, encapsulation, hierarchical organization, and polymorphism.

An object is something which contains data and does certain defined actions. It may be a particular AND gate, *ANDGate1*, for example. A class is the description of an object, *AND* for this example. It contains the description of the data structure of the object, and a set of procedures (called methods) which describe the behaviour of the object. In this example, the data structure would hold the parameters of the AND gate model peculiar to *ANDGate1*. An object is told to do something by passing it a message, *calculateNewOutput* for example. This message invokes one of the methods defined within the class *AND*.

The class structure is defined such that the data structures defined within a class cannot be altered, except by the methods defined within the class. This encapsulates the

data within an object, and does not allow any arbitrary changes to be made to that information, except by using one of the predefined message calls. This forces the programmer to design complete and unambiguous interfaces to each class of objects.

If two similar classes are to be defined, for example *AND* and *OR*, a great deal of their description will be common. Objective C allows class definitions to inherit parts of their definition from superclasses. Therefore all common parts of the class definitions of *AND* and *OR* can be grouped into one superclass *LogicElements*. The organization of subclasses and superclasses is called a class hierarchy. Much code development can be saved by putting as much common code into superclasses as possible.

The final saving in code development in the Objective C environment involves the concept of polymorphism, or late binding. In a program, object variables are not strongly typed. A variable can be assigned to be of any class type at run time. In sending a message to that variable, the run-time system checks to see what is the current class of the object variable, then calls up the method within that class's definition. By having a method in several class definitions, *calculateNewOutput* in the classes *AND* and *OR* for example, a variable which may be of type *AND* or *OR* at run time can be sent the message *calculateNewOutput* and the correct method will be implemented for the program at that time. Code size is reduced by leaving the choice of methods to execute up to run time parsing of message calls. No longer is it necessary to have a long if..then..else.. loop in a program every time a decision in an execution path has to be taken.

A.2 Definitions of Objective C and SMALLTALK Terms

instance variable - variables defined as part of a class definition.

method - definition of a message call within a class.

instance - instantiation of a class definition - an Objective C variable.

self - term used in a class definition to refer to an instance of that class. It is a convention in SMALLTALK to return self if nothing else is to be returned by a message.

id - new data type defined by Objective C. This is the generic type for any instance of a class, used like the C data types int and float. In effect, each class definition is a special definition of "id".

Appendix B ROOMMS Class Descriptions

This appendix is divided into two sections. The first quickly introduces each class implemented in ROOMMS and its significant features. The second consists of a detailed description of naming and format conventions and any storage constructs not discussed in the first part.

B.1 ROOMMS Class Descriptions

Figure 3-8 (page 53) shows all the classes implemented in ROOMMS. This section will give a brief description of each of those classes.

B.1.1 *Object*

The class *Object* in Figure 3-8 represents the Objective C class libraries. By definition, the class *Object* must be a superclass of all programs. (All Objective C classes are identified by italics in this appendix)

B.1.2 SimulationUtilities

The class `SimulationUtilities` is implemented as the superclass of `Simulation`, which is an instance of `ROOMMS`. In effect, `SimulationUtilities` is an extension of the `Simulation` class. `SimulationUtilities` contains the data definitions required for `Simulation`. Four data structures are defined as instance variables of the class `SimulationUtilities`. These are "elementNames", "elementNamesKeys", "nodeNames" and "nodeNamesKeys". These data structure are used throughout the `ROOMMS` program. They are created during the initialization phase of the simulator.

"elementNames" holds an *OrdCln* of all elements in the current simulation.

"elementNamesKeys" hold an *OrdCln* of the names of each of the elements in "elementNames", in the same order. "elementNamesKeys" is used as a key into "elementNames". The entries in "elementNamesKeys" are *String* representations of the names of the elements.

"nodeNames" holds a *OrdCln* of all the nodes contained in the circuit being simulated.

"nodeNamesKeys" holds an *OrdCln* of the names of each of the nodes in "nodeNames", in the same order. The entries are *String* representations. This instance variable is used the same as "elementNamesKeys"

Auxiliary instance variables are "numberOfElements", an integer, and "numberOfNodes", an integer.

This class implements a number of methods used as tools by the class `Simulation`.

method name	inputs	return (output)	discussion
atElementNamed:	aName (STR)	element	returns element with name "aName"
atNodeNamed:	aName (STR)	node	returns node with name "aName"
elementNamesAtKey:	aName (<i>String</i>)	element	returns element with name "aName"
elementNamesAtKey: put:	aName (<i>String</i>), anElement (Element)	element	puts anElement and key aName in elementNames and elementNamesKeys, updates numberOfElements
elementNamesAtName: put:	aName (STR), anElement (Element)	element	puts anElement and key aName as <i>String</i> in elementNames and elementNamesKeys, updates numberOfElements
initialize	-	instance of self	used to create new instance of Simulation - initializes instance variables
name	-	SimulationName (STR)	returns the name of the simulation
name:	aString (STR)	SimulationName (STR)	assigns name to simulation and returns the name of the simulation
nameOfElement:	anElement (Element)	name of element (STR)	returns string name of anElement
nameOfNode:	aNode (Node)	name of node (STR)	returns string name of aNode
nodeNamesAtKey: put:	aName (<i>String</i>), aNode (Node)	aNode (Node)	puts aNode and key aName in nodeNames and nodeNamesKeys, updates numberOfNodes
nodeNamesAtKey:	aName (<i>String</i>)	a node	returns a node corresponding to key aName
nodeNamesAtName: put	aName (STR), aNode (Node)	aNode	puts aNode and key corresponding to aName in nodeNames and nodeNamesKeys, updates numberOfNodes

method name	inputs	return (output)	discussion
nodeNamesIncludes:	aName (STR)	YES or NO	returns YES if node with name aName exists

B.1.3 Simulation

Simulation is implemented as a subclass of SimulationUtilities. Simulation is the main class of the simulator. The simulation is an instance of the class Simulation. Simulation inherits the data structures and methods defined within SimulationUtilities.

The following methods are defined within Simulation:

methods	input	returns (output)	discussion
example	-	-	initiates input, initiation and simulation phases of simulator
handleTimeStep:	aTime (int)	self	implements "Handle Timestep" process
initialize	-	-	initializes simulation variables, and simulator in general
makeFinalNode-Connections	-	self	reinitializes nodes, determines node types, creates element to node interconnections, and initializes some preprocessing for simulation
newElementOfClass:	aName (STR)	element	creates new element of class with name aName
output	-	self	prints out header information into OutputStream
simulation	-	-	implements "timestepper" process - executes simulation phase of simulator

2 OF 2

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

B.1.4 CircuitInput

This class is a subclass of *SimulationUtilities*, and is used to input a source file. It defines no instance variables. This class was written as a separate work [38]. This class was written in C, with a few key message calls which interact with the rest of the simulator.

This class implements one method:

methods	input	returns (output)	discussion
loadFrom:	filename (STR)	self	reads input file, loads essential data structures of simulator

B.1.5 Scheduler

The scheduler class implements the scheduler process of Figure 2-3. It is a subclass of *Object*. There is one instance of Scheduler in each simulation. It has three instance variables, "queue", "currentEvent", and "nextEvent".

"queue" is the simulation's scheduler queue. It is a *Dictionary*. The *Dictionary* contains entries which are evaluation queues (as *OrdCltns*) at specific timesteps.

"currentEvent" and "nextEvent" contain the Node which is the current and next event to be handled, respectively. They are used as pointers into "queue".

The following methods are defined within Scheduler:

methods	input	returns (output)	discussion
addSource: atTime:	anOutputNode (Node), nextTime (int)	self	places a source output node (anOutputNode) at the beginning of the queue entry at nextTime
addTime:	aTime (int)	new <i>OrdCln</i> of events to be handled at aTime	does error checking as necessary
add: toSchedulerQueue- AtTime:	aNode (Node), aTime (int)	self	adds aNode to evaluation queue at aTime
add: toSchedulerQueue- AtTime: withState:	aNode (Node), nextTime (int), aState (LogicState)	-	logic simulation scheduling algorithm. Would be more logically placed in logic nodes class, but complexity and interaction with scheduler queue means it is easier to implement here.
removeEmptyEntry- inQueueAtTime:	aTime (int)	self	removes an empty evaluation queue from queue
currentEvent:	-	returns currentEvent	
dropScheduledFol- OINode: at:	aNode (Node), aTime (int)	self	removes previously scheduled node from queue
getFirstEvent:	aTime (int)	a node	returns the first event in the evaluation queue at aTime
getNextEvent:	aTime (int)	a node	get next event at aTime or next event at later time
initialize	-	self	initializes instance variables
printSchedulerQueue	-	-	debugging routine - prints instance variables
atTime:	aTime (int)	evaluation queue (an <i>OrdCln</i>)	returns evaluation queue at aTime
reportGlitch: at: and: and:	aNode (Node), scheduled- Time (int), nextTime (int), lastTime (int)	GlitchFlag	increments GlitchFlag (for now)
rescheduleCurrent- EventAt:	atimestep (int)	self	put currentEvent at head of evaluation queue at atimestep

methods	input	returns (output)	discussion
timeIsInScheduler-Queue:	aTime (int)	YES or NO	returns YES if an entry exists in queue for aTime

B.1.6 SpecialDependencyObject

SpecialDependencyObject contains a class defined within SMALLTALK but not implemented in the class libraries of Objective C. This class is used to implement the interconnections between instances of Nodes and instances of Elements. It uses a global variable "Dependents" instead of instance variables.

SpecialDependencyObject implements the following methods:

methods	input	returns (output)	discussion
addDependent:	anObject(id)	self	adds an object as a dependent of self
allDependents	-	<i>OrdCln</i>	returns all of self's dependents in <i>OrdCln</i>
broadcast:	aSymbol (STR)	self	sends the argument, aSymbol, as a message to all self's dependents
broadcast: with:	aSymbol (STR), anObject (id)	self	sends the argument, aSymbol, as a message with the argument anObject to all self's dependents
changed	-	self	self has changed in some fashion - sends each dependent an update message
changed:	aParameter (id)	self	self has changed in some fashion - sends each dependent an update message

methods	input	returns (output)	discussion
changed: with:	firstParameter (id), secondParameter (id)	self	self has changed in some fashion - sends each dependent an update message with a parameter
dependents	-	<i>OrdCln</i>	returns <i>OrdCln</i> of dependents of self
dependsOn:	anObject (id)	self	adds self as dependent of anObject
initDependents	-	-	initializes the global variable Dependents
printDependents	-	-	debug routine - prints all of Dependents
release	-	self	removes all dependencies of self from Dependents
removeDependent:	anObject (id)	self	removes anObject from self's dependencies
update:	aParameter (id)	-	default method definition - does nothing
update: with:	firstArgument (id), secondArgument (id)	-	default method definition - does nothing

B.1.7 Nodes

This class is the superclass of all node classes. It defines the common instance variables and methods for all node classes. It defines the instance variables "scheduleTime", "fanouts", "fanins", "capacitance", "outputFlag", and "scheduleFlag".

"scheduleTime" is an integer containing the next timestep that the node is scheduled.

"fanouts" is an *OrdCln* of *String* representations of the names of the Elements which have this node as an output node.

"fanins" is an *OrdCln* of *String* representations of the names of the Elements which have this node as an input node.

"capacitance" is a float variable. It is used to hold a value of capacitance to ground loading on a node.

"outputFlag" is an integer variable used as a flag. It tells the particular node whether it is an output node, or not. "outputFlag" is also used to identify a "group" for analog nodes. A "group" is an integer number which will indicate to a graphical display program to display the "grouped" nodes together. For LogicNodes "outputFlag" is equal to 0 or 1, and the default for an output AnalogNodes is 1.

"scheduleFlag" is an integer variable. It is used during the Simulation Phase of the simulation in order to keep track whether the node has been scheduled, or not.

The following methods are defined within Nodes:

methods	input	returns (output)	discussion
capacitance	-	capacitance	returns the float value of capacitance
capacitance:	aCapacitance (float)	capacitance	assigns aCapacitance to capacitance and returns the float value capacitance
connections	-	<i>OrdCln</i>	returns an <i>OrdCln</i> of all the nodes connected to self
connectionsDelete:	aString (STR)	self	removes element named by aString from self's fanins and fanouts

methods	input	returns (output)	discussion
faninAdd:	anElement (id)	fanins	adds anElement to fanins and returns updated fanins
fanins	-	fanins	returns fanins
fanins:	aCollection (OrdCln)	fanins	replaces fanins with aCollection
fanoutAdd:	anElement (id)	fanouts	adds anElement to self's fanouts
fanouts	-	fanouts	returns fanouts
fanouts:	aCollection (OrdCln)	fanouts	replaces fanouts with aCollection
initialize	-	self	initializes instance variables
isOutputNode	-	YES or NO	returns YES if an output node
isScheduled	-	YES or NO	returns YES if node is scheduled
isSourceControlled-Node	-	NO	returns NO - all source controlled nodes will be instances of SourceLogicNode or SourceAnalogNode
isSourceOutputNode	-	NO	returns NO - all source controlled nodes will be instances of SourceLogicNode or SourceAnalogNode
name	-	name of node as STR	returns name of self as a C string
+new	-	self	creates new node
reinitialize	-	self	reinitializes scheduleTime
removeOutputFlag	-	self	turns of removeOutputFlag
schedulerFlag	-	scheduleFlag (int)	returns value of scheduleFlag
scheduleFlag:	aValue (int)	scheduleFlag (int)	sets scheduleFlag and returns its value
scheduleTime	-	scheduleTime (int)	returns scheduleTime
scheduleTime:	aTime (int)	scheduleTime (int)	resets scheduleTime to aTime and returns scheduleTime
setOutputFlag:	aFlag (int)	self	sets output flag with aFlag

B.1.8 LogicNodes

LogicNodes is a subclass of Nodes. In addition to the generic data structures defined in Nodes, LogicNodes have defined the instance variables "previousState" and "currentState".

"previousState" is an instance of the class LogicState. An instance of LogicState has two values stored within it, corresponding to a logic value (0 => logic low, 1 => logic high, and 2 => logic unknown), and a logic strength. Logic strengths refer to the driving, weak, or tri-state 'strengths', and correspond to the ability of an element to force its output value on a node. A logic element with a higher strength will 'win' a node. The range of strengths available to the simulation is determined by the user, and defaults to three. "previousState" holds the LogicState for the node which is valid up to one timestep before "now", unless "scheduleTime" is greater than "now" (meaning that the node is scheduled to become an event in the future). If the event is scheduled in the future, then previousState is valid now.

"currentState" is an instance of the class LogicState. "currentState" becomes valid at the timestep equal to "scheduleTime".

The following methods are defined in LogicNodes:

methods	input	returns (output)	discussion
currentState	-	currentState	returns current logic state of node
currentState:	newState (LogicState)	currentState	updates current logic state on node with newState

methods	input	returns (output)	discussion
currentVoltage	-	float	implicit logic to voltage conversion
currentVoltage:	aVoltage (float)	aVoltage	implicit voltage to logic conversion - assigns corresponding logic state of aVoltage to self
initialize	-	self	initializes instance variables of self
isAnalogNode	-	NO	returns NO
isLogicNode	-	YES	returns YES
output	-	self	prints state of node to OutputStream
previousState	-	previousState	returns previousState
previousState:	newState (LogicState)	newState	replaces previousState with newState
printOn:	anIOD (IOD)	self	debugging output routine
setInitialConditions:	aValue (float)	self	sets currentState to aValue, sets scheduleTime to 0
update:	anElement (id)	self	response to anElement sending the message changed to itself - implements logic algorithm

B.1.9 SLN (Source Logic Node)

This class is a subclass of *LogicNodes*. It is implemented as an input or output node of a source. It reacts just like any *LogicNode*, except in terms of scheduling. It is always scheduled at the head of an evaluation queue. An instance of SLN must have either zero or one source as an fanin. It may have any number of fanouts, sources included. It defines two instance variables, "sourceCollection", and "sourceTypeFlag".

"sourceCollection" is an *OrdCltn* of source elements. If "sourceTypeFlag" is YES, the last entry of "sourceCollection" is the source driving the SLN.

"sourceTypeFlag" is an integer flag and counter. If an SLN is a source output node, "sourceTypeFlag" equates to one less than the number of nodes in "sourceCollection".

SLN implements the following methods:

methods	input	returns (output)	discussion
currentState	-	currentState	returns currentState on node - going to source element if a source output node
initialize	-	self	initializes instance variables
isSourceControlled-Node	-	YES	returns YES
isSourceOutputNode	-	YES or NO	returns whether SLN is output of source or not
presentState	-	currentState	returns currentState to source element
printOn:	anIOD (IOD)	self	debugging printing routine
sourceCollection	-	sourceCollection	returns instance variable
sourceCollection:	aCollection (OrdCln)	sourceCollection	replaces sourceCollection with aCollection
sourceTypeFlag	-	sourceTypeFlag	returns integer value
sourceTypeFlag:	aNumber (int)	sourceTypeFlag	replaces sourceTypeFlag with aNumber
update:	aNode (id)	self	have become an event and must determine new value - sends message to source element that has self as output node to calculate a new value and reschedule the SLN. Also sends message to all sources that have SLN as input node that input node changed

B.1.10 AnalogNodes

AnalogNodes is a subclass of Nodes. In addition to the generic data structures defined in Nodes, AnalogNodes has defined the instance variables "previousTimeProcessed", "previousVoltage", "lastTimeProcessed", "currentVoltage", and "convergeFlag".

"previousTimeProcessed" is an integer variable used during the Simulation Phase of the simulator. It is defined as being the time (in timesteps) before the time that the node was last evaluated.

"previousVoltage" is a float variable used during the Simulation Phase of the simulator. It is defined as being the voltage on the node up to the timestep before the "lastTimeProcessed". In other words, it is the voltage on the node between the times "previousTimeProcessed" and "lastTimeProcessed" - 1, inclusive.

"lastTimeProcessed" is an integer variable used during the Simulation Phase of the simulator. It is defined as being the time (in timesteps) that the node was last evaluated. If the node is being processed now, then "lastTimeProcessed" is defined to be "now".

"currentVoltage" is a float variable used during the Simulation Phase of the simulator. It is defined as being the most recent voltage on the node. It corresponds to the voltage at the "lastTimeProcessed".

"convergeFlag" is an integer variable used as a flag during the Simulation Phase of the simulator. It is used to tell the node whether or not it has converged to a final value at a particular timestep.

The following methods are defined with AnalogNodes:

methods	input	returns (output)	discussion
convergeFlag	-	convergeFlag	returns an integer
convergeFlag:	aFlag (int)	convergeFlag	replaces convergeFlag with aFlag
currentState	-	aLogicState	implicit voltage to logic conversion
currentState:	aState (LogicState)	aState	implicit logic to voltage conversion - sets currentVoltage to voltage corresponding to aState
currentVoltage	-	currentVoltage	returns a float
currentVoltage:	aVoltage (float)	aVoltage	sets currentVoltage to aVoltage
initialize	-	self	initializes instance variables
isAnalogNode	-	YES	returns integer
isLogicNode	-	NO	returns integer
lastTimeProcessed	-	lastTimeProcessed	returns integer
lastTimeProcessed:	aTime (int)	lastTimeProcessed	replaces lastTimeProcessed with aTime
output	-	self	prints previous voltage of node to OutputStream
previousTime-Processed	-	previousTimeProcessed	returns integer instance variable
previousTime-Processed:	aTime (int)	previousTimeProcessed	replaces previousTimeProcessed with aTime
previousVoltage	-	previousVoltage	returns float instance variable
previousVoltage:	aVoltage (float)	previousVoltage	replaces previousVoltage with aVoltage
printOn:	anIOD (IOD)	self	debugging print routine
setInitialConditions:	aValue (float)	self	sets previousVoltage and currentVoltage to aValue and previousTimeProcessed and lastTimeProcessed to 0, and sets convergeFlag to YES

methods	input	returns (output)	discussion
update:	anElement (id)	self	response to an element sending itself the message "changed" - implements the Iterated Timing Analysis algorithm

B.1.11 SAN (Source Analog Node)

This class is a subclass of AnalogNodes. It is implemented as an input or output node of a source. It reacts just like any AnalogNode, except in terms of scheduling. It is always scheduled at the head of an evaluation queue. An instance of SAN must have either zero or one source as an fanin. It may have any number of fanouts, sources included. It defines two instance variables, "sourceCollection", and "sourceTypeFlag".

"sourceCollection" is an *OrdCln* of source elements. If "sourceTypeFlag" is YES, the last entry of "sourceCollection" is the source driving the SAN.

"sourceTypeFlag" is an integer flag and counter. If an SAN is a source output node, "sourceTypeFlag" equates to one less than the number of nodes in "sourceCollection".

SAN implements the following methods:

methods	input	returns (output)	discussion
currentVoltage	-	currentVoltage	returns currentVoltage on node - going to source element if a source output node
initialize	-	self	initializes instance variables

methods	input	returns (output)	discussion
isSourceControlled-Node	-	YES	returns YES
isSourceOutputNode	-	YES or NO	returns whether SAN is output of source or not
presentVoltage	-	currentVoltage	returns currentVoltage to source element
previousVoltage	-	previousVoltage	returns previousVoltage on node - going to source element if a source output node
printOn:	anIOD (IOD)	self	debugging printing routine
sourceCollection	-	sourceCollection	returns instance variable
sourceCollection:	aCollection (OrdCln)	sourceCollection	replaces sourceCollection with aCollection
sourceTypeFlag	-	sourceTypeFlag	returns integer value
sourceTypeFlag:	aNumber (int)	sourceTypeFlag	replaces sourceTypeFlag with aNumber
update:	aNode (id)	self	have become an event and must determine new value - sends message to source element that has self as output node to calculate a new value and reschedule the SAN. Also sends message to all sources that have SAN as input node that input node changed

B.1.12 Source

Source is a superclass for all source classes. It defines the common instance variables for all sources, "modelParameters", and "outputNode".

"modelParameters" is an *OrdCln* of the model parameters of the source. (See Appendix C).

"outputNode" contains the output node of the source, either an SAN or SLN.

Source defines the following methods:

methods	input	returns (output)	discussion
addConnection:	aNodeString (STR)	self	default method - works for TimedSources and ConstantSources only - sets outputNode to <i>String</i> representation of aNodeString
addToNodeLists	-	self	do nothing
createClockNode- From:	aNode (id)	self	default - do nothing
createInputNode- From:	aNode (id)	self	default - do nothing
createOutputNode- From:	aNode (id)	outputNode	default - do nothing
initialize	-	self	initialize modelParameters
isAnalogElement	-	NO	returns integer
isControlledSource	-	NO	returns integer
isLogicElement	-	NO	returns integer
modelParameters	-	modelParameters	returns <i>OrdCln</i>
modelParameters:	aCollection (<i>OrdCln</i>)	modelParameters	replaces modelParameters with aCollection
modelParametersAt: becomes:	anIndex (int), aNumber (float)	aNumber	assigns aNumber to entry anIndex in modelParameters
modelParameter- Sharing	-	-	do nothing
modelType	-	STR	returns name of the class of self
name	-	STR	returns name of self (i.e. the label of the source)
+new	-	self	creates a source
nodeConnections	-	<i>OrdCln</i>	default method - works only for sources with no input nodes - returns outputNode as sole entry in collection

methods	input	returns (output)	discussion
nodeConnections:	aCollection (id)	self	default method - works only for sources with no inputs - assigns first entry in aCollection to: outputNode
numberOfConnections	-	1	default method - true for sources with no inputs - returns integer 1
outputNode	-	outputNode	returns SAN or SLN
outputNode:	aNode (id)	outputNode	replaces outputNode with aNode
parameterCheck	-	YES	must be implemented in subclasses
printOn	anIOD (IOD)	self	debugging printing routine
update	-	self	do nothing

B.1.13 ControlledSource

ControlledSource is a superclass of all controlled sources. It is a subclass of Source and inherits Source's instance variables and methods. ControlledSource defines one instance variable, "inputNode".

"inputNode" contains the input node of the controlled source, either an SAN or SLN.

ControlledSource implements the following methods:

methods	input	returns (output)	discussion
addConnection:	aNodeString (STR)	self	do nothing

methods	input	returns (output)	discussion
createInputNode-From:	aNode (id)	inputNode	default method works for controlled sources which have an analog input node - assigns aNode as inputNode, creating a new SAN, if necessary
createOutputNode-From:	aNode (id)	outputNode	default method works for controlled sources which have an analog output node - assigns aNode as outputNode, creating a new SAN if necessary
inputNode	-	inputNode	returns instance variable
inputNode:	aNode (id)	inputNode	replaces inputNode with aNode
isControlledSource	-	YES	returns integer
nodeConnections	-	IntArray	returns outputNode and InputNode - used only during set-up when nodeConnections would be expected to be an array of numbers, not nodes
nodeConnections:	aCollection (id)	self	replaces outputNode and inputNode with contents of aCollection
numberOfConnections	-	2	returns integer 2 - must be redefined for LcvcvS
printOn:	anIOD (IOD)	self	debugging printing routine

B.1.14 LcvS (Logic Controlled Voltage Source)

LcvS is a subclass of ControlledSource, and contains the definitions of one type of source element. LcvS defines two instance variables, "transition", and "timeOfTransition".

"transition" contains an integer flag which contains the information of what transition the input signal has gone through. This is required for determining the proper delay.

"timeOfTransition", an integer, contains the time when the input signal changed.

Lcvs implements the following methods:

methods	input	returns (output)	discussion
createInputNode-From:	aNode (id)	inputNode	assigns aNode as inputNode, creating a new SLN if necessary
currentVoltage	-	currentVoltage	returns float
inputNodeChanged	-	self	inputNode has changed, and want to schedule output node
initialize	-	self	initializes instance variables
parameterCheck	-	YES	checks modelParameters, converts all float values into integers.
printOn:	anIOD (IOD)	self	debugging printing routine
scheduleUpdate	-	self	updates output node and reschedules it.

B.1.15 Vcls (Voltage Controlled Logic Source)

Vcls is a subclass of ControlledSource, and contains the definitions of one type of source element. Vcls defines no instance variables.

Vcls implements the following methods:

methods	input	returns (output)	discussion
createOutputNode-From:	aNode (id)	outputNode	assigns aNode as outputNode, creating a new SLN if necessary
currentState	-	currentState	does explicit voltage to logic conversion from inputNode's currentVoltage
inputNodeChanged	-	self	inputNode has changed, and want to schedule output node
output	-	self	output routine - prints currentState to OutputStream
parameterCheck	-	YES	checks modelParameters
printOn:	anIOD (IOD)	self	debugging printing routine
scheduleUpdate	-	self	updates output node and reschedules it.

B.1.16 Vcvs (Voltage Controlled Voltage Source)

Vcvs is a subclass of ControlledSource, and contains the definitions of one type of source element. Vcvs defines instance variables, "delay", "triggerTimes", "inputVoltages", "lastTriggerTime", and "lastInputVoltage".

"delay" contains an integer which holds the delay parameter of the source.

"triggerTimes", an *OrdCln*, contains the times when the input node changed between now and "delay" timesteps ago.

"inputVoltages", an *OrdCln*, contains the input voltage when the node changed between now and "delay" timesteps ago.

"lastTriggerTime", an integer, contains timestep when input node last changed.

"lastInputVoltage", a float, contains the input node's currentVoltage at "lastTriggerTime".

Vcvs implements the following methods:

methods	input	returns (output)	discussion
currentVoltage	-	float	returns presentVoltage of output node
initialize	-	self	initializes instance variables
inputNodeChanged	-	self	inputNode has changed, and want to schedule output node
parameterCheck	-	YES	checks modelParameters, assigns delay parameter to delay instance variable, and replaces modelParameters with just the coefficients
printOn:	anIOD (IOD)	self	debugging printing routine
scheduleUpdate	-	self	updates output node and reschedules it.

B.1.17 Lcvcvs (Logic Clocked Voltage Controlled Voltage Source)

Lcvcvs is a subclass of Vcvs, and contains the definitions of one type of source element. The behaviour of Lcvcvs is very similar to Vcvs, so it inherits its definitions. Lcvcvs defines two instance variables, "clockNode", and "presentClockValue".

"clockNode" contains an SLN, the logic input node.

"presentClockValue", an integer, contains the logic value on "clockNode".

Lcvcvs implements the following methods:

methods	input	returns (output)	discussion
clockNode	-	clockNode	returns instance variable
clockNode:	aNode (id)	clockNode	replaces clockNode with aNode
createClockNode-From:	aNode (id)	clockNode	assigns clockNode to aNode, creating a SLN, if required
initialize	-	self	initializes instance variables
inputNodeChanged	-	self	one of two input nodes has changed significantly - reschedule output node at next appropriate timestep
nodeConnections	-	<i>IntArray</i>	returns an array of node connections - used only during set-up
nodeConnections:	aCollection (id)	self	replaces outputNode, inputNode and clockNode with contents of aCollection
numberOf-Connections	-	3	returns the integer 3
printOn:	anIOD (IOD)	self	debugging printing routine

B.1.18 TimedSource

TimedSource is a superclass of all timed sources. A timed source drives a node with a time dependent signal. TimedSource is a subclass of Source and inherits Source's instance variables and methods. TimedSource does not define any instance variables.

TimedSource implements the following methods:

methods	input	returns (output)	discussion
createOutputNode-From:	aNode (id)	outputNode	default method works for timed sources which have an analog output node - assigns aNode as outputNode, creating a new SAN if necessary
parameterCheck	-	the integer "1"	Verifies model parameters. Converts all units of time to timesteps. Removes superfluous parameters.

B.1.19 Lsrc (Logic Timed Source)

Lsrc is a subclass of TimedSource, and contains the definitions of one type of source element. Lsrc does not define any instance variables. Lsrc is used to create a pulse logic signal.

Lsrc implements the following methods:

methods	input	returns (output)	discussion
createOutputNode-From:	aNode (id)	outputNode	assigns aNode as outputNode, creating a new SLN if necessary
currentState	-	presentState on outputNode	calculates and determines new output state based on time and model parameters.
output	-	self	special output routine for Lsrc - required for vertical pulses (ie without rise and fall delays)
printOn:	anIOD (IOD)	self	debugging printing routine
scheduleUpdate	-	self	calculates new output, updates output node and reschedules it.

B.1.20 Tsrc (Analog Timed Source)

Tsrc is a subclass of TimedSource, and contains the definitions of one type of source element. Tsrc defines no instance variables. Tsrc implements an analog pulse waveform.

Tsrc implements the following methods:

methods	input	returns (output)	discussion
currentVoltage	-	presentVoltage on outputNode	returns presentVoltage on outputNode (a float)
printOn:	anIOD (IOD)	self	debugging printing routine
scheduleUpdate	-	self	calculates new output, updates output node and reschedules it.

B.1.21 Sinusoid (Sinusoid Timed Source)

Sinusoid is a subclass of TimedSource, and contains the definitions of one type of source element. Sinusoid implements an analog sinusoid waveform. Sinusoid defines one instance variable, "schedulingPeriod".

"schedulingPeriod" contains an integer which holds the minimum number of timesteps in which the sinusoid output changes significantly.

Sinusoid implements the following methods:

methods	input	returns (output)	discussion
currentVoltage	-	float	calculates and returns currentVoltage of output node
initialize	-	self	initializes instance variables
parameterCheck	-	YES	checks modelParameters, converts time units to timestep units (Note period rounded off to nearest timestep), removes superfluous parameters, calculates schedulingPeriod
printOn:	anIOD (IOD)	self	debugging printing routine
scheduleUpdate	-	self	reschedules outputNode every schedulingPeriod

B.1.22 General Source (An Analog Timed Source)

GeneralSource is a subclass of TimedSource, and contains the definitions of one type of source element. Outputs an analog signal in the form of a time dependent polynomial. GeneralSource does not define any additional instance variables.

GeneralSource implements the following methods:

methods	input	returns (output)	discussion
currentVoltage	-	aVoltage (float)	calculates a new output voltage, assigns it to outputNode
parameterCheck	-	YES	checks model parameters
printOn:	anIOD (IOD)	self	debugging printing routine
scheduleUpdate	-	self	cheats - schedules GeneralSource outputNode at next timestep (i.e. every timestep)

B.1.23 **ConstantSource**

ConstantSource is a superclass of all constant sources. It is a subclass of Source and inherits Source's instance variables and methods. ConstantSource does not define any instance variables, or methods. This class is a place holder in the class hierarchy for the two types of constant signal sources, Lrail and Vrail.

B.1.24 **Lrail (Logic Rail - Constant Logic Source)**

Lrail is a subclass of ConstantSource, and contains the definitions of one type of source element. Lrail defines one instance variable, "constantState". Lrail generates a constant logic state onto its output node.

"constantState" contains the only model parameter required of this source element, a logic state.

Lrail implements the following methods:

methods	input	returns (output)	discussion
createOutputNode-From:	aNode (id)	outputNode	assigns aNode as outputNode, creating a new SLN if necessary
currentState	-	currentState	returns the value of constantState
initialize	-	self	initializes instance variables
parameterCheck	-	YES	checks modelParameters, assigns value to constantState

methods	input	returns (output)	discussion
printOn:	anIOD (IOD)	self	debugging printing routine
scheduleUpdate	-	self	does nothing, other than to verify that outputNode is at constantState

B.1.25 Vrail (Voltage Rail Constant Analog Source)

Vrail is a subclass of ConstantSource, and contains the definitions of one type of source element. Vrail defines one instance variable, "DCVoltage".

"DCVoltage" is a float value containing the Vrail's constant output voltage.

Vrail implements the following methods:

methods	input	returns (output)	discussion
createOutputNode-From:	aNode (id)	outputNode	assigns aNode as outputNode, creating a new SAN if necessary
currentVoltage	-	DCVoltage	returns the constant voltage output, DCVoltage
initialize	-	self	initializes instance variables
parameterCheck	-	YES	checks modelParameter, sets DCVoltage
printOn:	anIOD (IOD)	self	debugging printing routine
scheduleUpdate	-	self	does nothing, except verify that outputNode contains DCVoltage

B.1.26 Elements

Elements is a superclass of all elements, implementing the common features of all logic and analog pre-defined element models. Elements defines two instance variables, "modelParameters", and "nodeConnections".

"modelParameters" is an *OrdCltm* of the model parameters of the particular element. (See Appendix C for definitions of model parameters for all element models).

"nodeConnections" is an *OrdCltm* of the nodes connected to the element in the order of outputs, input/outputs, inputs (at least during simulation phase of simulation. During set-up phase, this is not necessarily true) during set-up phase, and an *IntArray* of offsets of the node in SimulationUtilities' nodeName during the simulation.

Elements implements the following methods:

methods	input	returns (output)	discussion
addConnection:	aNodeString (STR)	self	adds aNodeString to nodeConnections (Note that nodeConnections will be changed at end of setup phase - see "nodeConnections:")
addToNodeLists	-	self	adds the element to its connected nodes' fanin and fanout lists, as appropriate.
initialize	-	self	initializes instance variables
isAnalogElement	-	NO	default method - redefine in subclass if necessary
isLogicElement	-	NO	default method - redefine in subclass if necessary

methods	input	returns (output)	discussion
modelParameters	-	<i>OrdCln</i> of modelParameters	returns modelParameters
modelParameters:	aCollection (id)	modelParameters (<i>OrdCln</i>)	assigns aCollection to modelParameters
modelParametersAt: anIndex: becomes:	anIndex (int), aNumber (float)	aNumber	assigns <i>String</i> representation of aNumber into the anIndex position of modelParameters
modelParameters- Labels	-	-	unused method
modelParameter- Sharing	-	-	unused method
name	-	(STR)	returns name of the element that is "self" as a C string
+new	-	self	creates new element
nodeConnections	-	nodeConnections	returns nodeConnections
nodeConnections:	aCollection (id)	self	Assigns the element's connectivity, as an <i>IntArray</i> , checks the connections, adds "self" to proper nodes' fanin and fanout lists
parameterCheck	-	-	default - does absolutely nothing
printOn:	anIOD (IOD)	self	debugging printing routine
update:	aNode (id)	self	the response to a Node sending itself the message "changed". The fanouts of the Node receive the message, and in turn tell their output nodes that the state on the output node may change.

B.1.27 AnalogElements

AnalogElements is a subclass of Elements, a superclass for all built-in analog elements models. AnalogElements defines the common behaviour of all analog elements that is not defined in Elements. AnalogElements does not define any instance variables.

Subclasses of AnalogElements will define the method "calculateIncrementsWith: becauseOf:", and if necessary, "nodeCheck" and "parameterCheck"

AnalogElements implements the following methods:

methods	input	returns (output)	discussion
addToFanoutsLists	-	self	adds "self" to all input and input/output nodes' fanout lists
calculateIncrement- With: becauseOf:	Vguess (float), aNode (id)	a new Increment value (id)	default method which returns an increment of conductance, current and capacitance generated by an element in an Increment. Redefined in all subclasses
inputCapacitance:	aNode (id)	0.0 (float)	Those analog elements with an input capacitance to ground will have to redefine this method.
isAnalogElement	-	YES	returns integer
isLogicElement	-	NO	returns integer
nodeCheck	-	an integer	default method, returns whether or not there are two nodes connected to element
numberOfOutput- Nodes	-	2 (integer)	default method - element has two output or input/output nodes
outputCapacitance:	aNode (id)	0.0 (float)	Those analog elements with an output capacitance to ground will have to redefine this method.

B.1.28 Fcapr (Floating Capacitor)

Fcapr is a subclass of AnalogElements, and contains the definitions of a floating capacitor. Fcapr defines no instance variables.

Fcapr implements the following method:

methods	input	returns (output)	discussion
calculateIncrement- With: becauseOf:	Vguess (float), aNode (id)	anIncrement	returns an increment of conductance, capacitance and current to aNode

B.1.29 Tres (Resistor)

Tres is a subclass of AnalogElements, and contains the definitions of an analog resistor. Tres defines no instance variables.

Tres implements the following method:

methods	input	returns (output)	discussion
calculateIncrement- With: becauseOf:	Vguess (float), aNode (id)	anIncrement	returns an increment of conductance, capacitance and current to aNode

B.1.30 MOSFET

MOSFET is a subclass of AnalogElements, superclass to Nmos and Pmos MOSFET transistors, and contains the common behaviour definitions of MOSFET transistors. MOSFET defines three instance variables, "partVt", "linKP", and "satKP".

"partVt" is a portion of threshold voltage which can be precalculated during the setup phase of the simulation.

"linKP" is KP multiplied by the width to length ratio of the transistor. This number is precalculated during the setup phase of the simulation.

"satKP" is linKP divided by 2. This number is precalculated during the setup phase of the simulation.

MOSFET implements the following methods:

methods	input	returns (output)	discussion
calculateIncrement- With: becauseOf:	Vguess (float), aNode (id)	anIncrement	returns an increment of conductance, capacitance and current to aNode (Note this is a common method with Pmos and Nmos)
nodeCheck	-	YES or NO	re-arranges order of nodeConnections during setup phase to drain, source, gate, bulk
parameterCheck	-	-	precalculates the intermediate products and stores them into the instance variables

B.1.31 Nmos (N-channel MOSFET)

Nmos is a subclass of MOSFET, and contains the definitions of an n-channel MOSFET. Nmos defines no instance variables, and implements only one method.

Nmos implements the following method:

methods	input	returns (output)	discussion
mosType	-	aFloat	returns 1.0 if Nmos, -1.0 if Pmos

B.1.32 Pmos (P-channel MOSFET)

Pmos is a subclass of MOSFET, and contains the definitions of an p-channel MOSFET. Pmos defines no instance variables, and implements only one method.

Pmos implements the following method:

methods	input	returns (output)	discussion
mosType	-	aFloat	returns -1.0 if Pmos, 1.0 if Nmos

B.1.33 Isourceelement (Constant Current Source)

Isourceelement is a subclass of AnalogElements, and contains the definition of a constant current source. It allows the addition of a constant current to an analog node. This source acts like any other analog element by adding a conductance and current value to an analog during the Iterated Timing Analysis algorithm. It does not drive a node like the rest of the sources.

Isourceelement implements the following methods:

methods	input	returns (output)	discussion
calculateIncrement- With: becauseOf:	Vguess (float), aNode (id)	anIncrement	Returns conductance and current values from modelParameters, and 0 capacitance
nodeCheck	-	YES or NO	checks to see that there is only one output node and no input node

methods	input	returns (output)	discussion
numberOfOutputNodes	-	1 (integer)	only one output node

B.1.34 Gcapr

Gcapr is a subclass of Elements. Gcapr is a special form of element, which behaves differently from all other elements. Gcapr elements represent a capacitor connected from a node to ground. The node data structure, itself, contains an instance variable which holds the total capacitance to ground from the node. Gcapr elements are preprocessed before simulation and the capacitance value of the Gcapr is added to the capacitance value on a Node, and the Gcapr element is ignored for the rest of the simulation. Gcapr elements behave the same for both analog and logic simulation modes. Gcapr does not define any instance variables.

Gcapr implements the following methods:

methods	input	returns (output)	discussion
addToFanoutsLists	-	self	only one node - Note used only during setup phase
calculateFallTime- With:	capacitance (float)	0 (int)	returns zero, since not a "true" element
calculateNewOutput- For:	aNode (id)	self	do nothing, not a "true" element
calculateRiseTime- With:	capacitance (float)	-	do nothing, not a "true" element
inputCapacitance:	aNode (id)	float	returns the Gcapr's contribution to the node's capacitance

methods	input	returns (output)	discussion
modelParameters-Labels	-	-	unused method
nodeCheck	-	YES or NO	checks to see if only one model parameter
numberOfOutput-Nodes	-	0.0 (integer)	returns 0, there are no output nodes for a Gcapr. By definition, Gcapr only has an input node as far as this is concerned
outputCapacitance:	aNode (id)	0.0 (float)	returns 0.0, there are no output nodes for a Gcapr, as far as this is concerned
update:	aNode (id)	self	does nothing, not a "true" element

B.1.35 LogicElements

LogicElements is a subclass of Elements, a superclass for all built-in logic element models. LogicElements defines the common behaviour of all logic elements that is not defined in Elements. LogicElements does not define any instance variables.

LogicElements implements the following methods:

methods	input	returns (output)	discussion
addToFanoutsLists	-	self	adds "self" to all input nodes' fanout lists
biasValue:	newValue (integer)	newValue (integer)	default method which is redefined in all subclasses which must invert a logic value
calculateFallTime-With:	capacitance (float)	fall time as integer	calculates fall time as units of timesteps
calculateNewOutput-For:	aNode (id)	a new logic state (id)	standard calculating routine for new output state using look-up table.

methods	input	returns (output)	discussion
calculateRiseTime- With:	capacitance (float)	rise time as integer	calculates rise time as units of timesteps
inputCapacitance:	aNode (id)	input capacitance of element as a float	returns input capacitance parameter of element
invertValue:	value (integer)	returns logic value as integer	inverts logic value
isAnalogElement	-	NO	returns integer
isLogicElement	-	YES	returns integer
nodeCheck	-	an integer	default method, returns whether or not there are a minimum of three nodes connected to element
numberOfOutput- Nodes	-	1 (integer)	default method - most logic elements have one output node
outputCapacitance:	aNode (id)	float	returns model parameter for output node capacitance

B.1.36 And

And is a subclass of LogicElements, and contains the definitions of an And gate.

And defines no instance variables.

And implements the following method:

methods	input	returns (output)	discussion
logicTable:	table (3x3 integer array)	self	assigns lookup table to the integer array "table"

B.1.37 Nand

Nand is a subclass of And, and contains the definitions of an Nand gate which are different from an And gate. Nand defines no instance variables.

Nand implements the following method:

methods	input	returns (output)	discussion
biasValue:	value (integer)	integer	inverts the logic value "value" (which contains the output of an And gate)

B.1.38 Or

Or is a subclass of LogicElements, and contains the definitions of an Or gate. Or defines no instance variables.

Or implements the following method:

methods	input	returns (output)	discussion
logicTable:	table (3x3 integer array)	self	assigns lookup table to the integer array "table"

B.1.39 Nor

Nor is a subclass of Or, and contains the definitions of an Nor gate which are different from an Or gate. Nor defines no instance variables.

Nor implements the following method:

methods	input	returns (output)	discussion
biasValue:	value (integer)	integer	inverts the logic value "value" (which contains the output of an And gate)

B.1.40 Xor

Xor is a subclass of LogicElements, and contains the definitions of an Xor gate. Xor defines no instance variables.

Xor implements the following method:

methods	input	returns (output)	discussion
logicTable:	table (3x3 integer array)	self	assigns lookup table to the integer array "table"

B.1.41 Xnor

Xnor is a subclass of Xor, and contains the definitions of an Xnor gate which are different from an Xor gate. Xnor defines no instance variables.

Xnor implements the following method:

methods	input	returns (output)	discussion
biasValue:	value (integer)	integer	inverts the logic value "value" (which contains the output of an And gate)

B.1.42 Inverter

Inverter is a subclass of LogicElements and contains the definition of an inverter. Inverter defines no instance variables.

Inverter implements the following methods:

methods	input	returns (output)	discussion
calculateNewOutput-For:	anotherNode (id)	a LogicState	returns the updated output state of the inverter - note does not use look-up table
nodeCheck	-	YES or NO	makes sure that inverter has only two node connections

B.1.43 Delay

Delay is a subclass of Inverter and contains the definition of an delay gate.

Implements a set delay in seconds. Delay defines no instance variables.

Delay implements the following methods:

methods	input	returns (output)	discussion
calculateFallTime- With:	capacitance (float)	float	returns the model parameter (only one)
calculateNewOutput- For:	anotherNode (id)	a LogicState	returns input node logic state after a set delay
calculateRiseTime- With:	capacitance (float)	float	returns the model parameter (only one)
inputCapacitance:	aNode (id)	0.0 as float	element has no input capacitance
outputCapacitance:	aNode (id)	0.0 as float	element has no output capacitance

B.1.44 Buffer

Buffer is a subclass of Inverter, and contains the definitions of an Buffer which are different from an Inverter. Buffer defines no instance variables.

Buffer implements the following method:

methods	input	returns (output)	discussion
invertValue:	aValue (integer)	aValue	doesn't invert the value, otherwise reacts just like an Inverter

B.1.45 FlipFlops

FlipFlops is a subclass of LogicElements and contains the common definitions of the flip flops. FlipFlops defines three instance variables, "transitions", "timeOfTranslations", and "delay".

FlipFlops implements the following methods:

methods	input	returns (output)	discussion
calculateFallTime- With:	capacitance (float)	float	returns the proper delay - for set, reset, enable, returns tpw+tcq - for clock, returns tcq
calculateRiseTime- With:	capacitance (float)	float	returns the model parameter (only one)
initialize	-	self	initializes instance variables
numberOfOutput- Nodes	-	2	flipflops have two output nodes - returns an integer
outputCapacitance:	aNode (id)	float	returns model parameter

B.1.46 Dflipflop

Dflipflop is a subclass of FlipFlops and contains the definition of the edge-triggered clocked, set-reset-enable D flip flop. Dflipflop does not define any instance variables.

Dflipflop implements the following methods:

methods	input	returns (output)	discussion
calculateNewOutput-For:	. Node (id)	new output logicstate	calculates new output of flip flop - difficult because of edge-triggering
initialize	-	self	initializes instance variables
inputCapacitance:	aNode (id)	float	returns input capacitance for aNode input
nodeCheck	-	YES or NO	Makes sure mandatory nodes are connected.

B.1.47 Jkflipflop

Jkflipflop is a subclass of FlipFlops and contains the definition of the edge-triggered clocked, set-reset-enable JK flip flop. Jkflipflop does not define any instance variables.

Jkflipflop implements the following methods:

methods	input	returns (output)	discussion
calculateNewOutput-For:	aNode (id)	new output logicstate	calculates new output of flip flop - difficult because of edge-triggering
initialize	-	self	initializes instance variables
inputCapacitance:	aNode (id)	float	returns input capacitance for aNode input
nodeCheck	-	YES or NO	Makes sure mandatory nodes are connected.

B.1.48 Increments

Increments is a subclass of Object. It is used to contain an increment of conductance, current and capacitance during a calculation of an analog node. Note that Increments are always used repeatedly during the most critical loop of the simulator, and this class should be replaced with a "C" record structure. This simple improvement will significantly reduce the message call overhead and is predicted to have a significant affect on the program's performance. Increments defines three instance variables, "conductance", "current", and "capacitance".

"conductance" contains an increment of the conductance supplied to an analog node by an analog element.

"current" contains an increment of the current supplied to an analog node by an analog element.

"capacitance" contains an increment of the capacitance supplied to an analog node by an analog element.

Increments defines the following methods:

methods	input	returns (output)	discussion
capacitance	-	float	returns capacitance
capacitance:	aCapacitance (float)	float	assigns capacitance value to instance variable
conductance	-	float	returns conductance
conductance:	aConductance (float)	float	assigns conductance value to instance variable
current	-	float	returns current
current:	aCurrent (float)	float	assigns current value to instance variable
+new	-	an Increment	creates new Increment and initializes it
printOn:	anIOD (IOD)	self	debugging output routine

B.1.49 LogicState

LogicState is a subclass of Object. It is used to contain a logic state, which consists of a logic value and a logic strength. Note that LogicStates are always used repeatedly during the most critical loop of the simulator, and this class should be replaced with a "C" record structure. This simple improvement will significantly reduce the message call overhead and is predicted to have a significant affect on the program's performance. LogicState defines two instance variables, "value", and "strength".

"value" contains the logic value, 0 for logic low, 1 for logic high, and 2 for logic unknown.

"strength" contains a logic strength.

LogicState defines the following methods:

methods	input	returns (output)	discussion
logicValue	-	integer	returns value
logicValue:	newValue (integer)	integer	assigns logic value to instance variable
+new	-	a new LogicState	creates a new LogicState and initializes the instance variables
printOn:	anIOD (IOD)	self	debugging output routine
strength	-	integer	returns strength
strength:	newStrength (integer)	integer	assigns new strength to instance variable

B.2 Naming Conventions, etc.

This section is intended to describe the naming and other format conventions adopted as part of the ROOMMS simulator.

B.2.1 Element Model Names

Element model names have the following format:

- first letter capitalized, rest lower case
- one word
- use abbreviations used by SPLICE [12]

B.2.2 Order of Nodes (with respect to the internal storage of Elements)

The internal connections between elements are made using nodes. Nodes are referenced within the data structure of an Element in the following order:

- Output nodes first, then input/output nodes, then input nodes

Output nodes are the unidirectional output connection of an element. Input/output nodes are bidirectional connections of an element. Input nodes are the unidirectional input connections of an element. Each element model will know the orientations of the nodes to which it is connected. Most element models will use a combination of the three orientations.

B.2.3 Determination of the Type of a Node

Node connections between elements are of one of the following types: Analog; Logic, SLN or SAN. The particular type of a node is determined during the Input Phase of the simulator. By convention:

- Input or Output nodes of a source are either SLNs or SANs;
- Output nodes of an analog element are AnalogNodes;
- Output nodes of a logic element are LogicNodes; and
- Conflicts are solved in favour of source nodes, then AnalogNodes.

B.2.4 Program Variable Names

Variables used within the program have been named, using the following convention:

- Names are as descriptive as possible
- Multiple word names are concatenated together, with the first word's first character in lower case, and the first letter of the subsequent words in upper case. Otherwise, all characters are in lower case, with few exceptions.
- Global variables follow the above rules, except that the first character of the first word of the name is upper case.
- Class names follow the rules for global variables.

B.2.5 Global Variables

There are a few key global variables. These are:

Global Variable	explanation
PresentTimeStep	integer containing the number of the timestep corresponding to "now"
DebugFlag	YES or NO - debugging flag - unused
GlitchFlag	integer which indicates whether glitches are to be reported to the user - set by Simulation Parameter card in input deck

Global Variable	explanation
NumberOfLogicStrengths	integer which indicates number of available logic strengths. Default and minimum is 3. Set by Simulation Parameter card.
StatisticsFlag	integer flag - unused
EndOfSimulation	integer containing the number of timestep which is the end of the simulation
CleanOutput	integer flag which modifies the output into a useful output - set by "-c" parameter on command line
StartTime	integer containing timestep when simulation starts - can be set to other than zero by Simulation Parameter card
Mrt	float containing timestep size in seconds - set by Simulation Parameter card
StopTime	float containing end of simulation in seconds - set by Simulation Parameter card
Sor	Successive Over-Relaxation parameter, a float - set by Simulation Parameter card.
RelativeTolerance	relative error tolerance for analog simulation, a float. Can be set by Simulation Parameter card
AbsoluteTolerance	absolute error tolerance for analog simulation, a float. Can be set by Simulation Parameter card
MinimumDeltaV	float variable which is the minimum voltage change on an analog node before it will be scheduled. Can be set by Simulation Parameter Card
LogicLevels	float array of two elements which gives values for implicit voltage to logic conversions. Can be set by Simulation Parameter card.
VoltageLevels	float array of three elements which gives values for implicit logic to voltage conversions. Can be set by Simulation Parameter card.
Dependents	a <i>Dictionary</i> used by SpecialDependencyObject. Used to hold all connections between nodes and elements.
SchedulerQueue	The instance of Scheduler class.
TheSimulation	The simulation - and instance of the Simulation class
SimulationName	string which contains the simulation name from the first line of source deck
InputReference	string which contains name of input file

Global Variable	explanation
OutputReference	string which contains name of output file
OutputStream	file pointer to output file

Appendix C ROOMMS User Guide

C.1 General

This appendix is separated into sections describing the initiation of a ROOMMS simulation, the format of a source deck, the ROOMMS logic element types, the ROOMMS analog element types, the ROOMMS source element types and a description of the format of the output file of ROOMMS.

C.2 Using ROOMMS

ROOMMS accepts three parameters: "-c", "input file name", and "output file name". "Input file name" and "output file name" must be in that order. "-c" is required to generate a SPLICE1.7 compatible output file for use with graphic display generators designed for that program. An example follows:

```
roomms -c inputfile outputfile
```

Each parameter is optional, although the first non "-c" parameter is assumed to be the input file name, and the second non "-c" parameter is assumed to be the output file name. The default input file name is "default.in", and the default output is "stdout" (i.e. to the screen).

C.3 ROOMMS Source Deck Format

The ROOMMS source deck format was modelled on the source deck formats of SPLICE1.7 [12]. However, the error checking of the SPLICE1.7 preprocessor was not implemented. The ROOMMS input deck must be correct. The intent was to use the ROOMMS input source deck as an interim file between a more comprehensive circuit parsing program and the ROOMMS simulator itself. Whitespace is generally ignored, but otherwise, all ordering requirements must be exactly followed.

Figure 2-2 is reproduced here as a reference.

Simple Circuit Example Source Deck

Model Cards

```
.model Rmodel res: 10000  
.model Cmodel fcapr: 10e-12  
.model vmodel sinusoid: 2.5 0.0 0 5.0e-8
```

Element Cards

```
.element v-element a-node c-node vmodel  
.element R-element a-node b-node Rmodel  
.element C-element b-node c-node Cmodel
```

Simulation Parameters

```
.time 5.0e-9 150.0e-9
```

Output Requests

```
a-node b-node c-node
```

Figure 2-2
Simple Circuit Problem
Example Source Deck

C.3.1 Simulation Title

The first line of the source deck is the simulation title. Any alphanumeric string on the first line of the source deck up to 80 characters long is acceptable.

C.3.2 Comments and Whitespace

Any line starting with a semi-colon is considered to be a comment. The general format of a comment is a semi-colon followed by a space, and then the comment:

`; Any_comment_whatsoever`

Comments are ignored.

In addition, all blank lines are ignored. Spaces, commas and colons can be added liberally within any model card, element card, simulation parameter card or output request card. As an example, the colons in the model cards of Figure 2-2 are unneeded, and are used only to visibly separate the model parameters from the ROOMMS element type entries on the model cards.

C.3.3 Model Cards

All model cards must be placed following the key line that follows:

Model Cards

Model cards are of the general form:

.model model_name ROOMMS_element_type : element_parameters

where: **.model** is a keyword which must start each model card.

model_name is any alphanumeric string, unique with respect to all other **model_names**. This is used to identify the particular element model.

ROOMMS_element_type is a predefined ROOMMS element type. Acceptable types are listed below. Upper or lower case characters are permitted.

element_parameters are a list of parameters defined by the ROOMMS element type. There are generally default values, but the order defined below is important. No parsing of this parameter list is done by ROOMMS. All entries must be float or integer values, as required by the ROOMMS element type. Extra parameters will be ignored.

A separate model card must be defined for each different element model since there is no way to modify an element with parameters on the element cards as in SPLICE1.7 [12] and SPICE [42].

C.3.4 Element Cards

All element cards must be placed following the key line that follows:

Element Cards

Element cards are of the general form:

.element element_name node_name(s) model_name

where: **.element** is a keyword, which must start each element card.

element_name is any alphanumeric string, unique with respect to all other **element_names**. This is used to identify the particular element.

node_name(s) are one or more node names, which can be any alphanumeric string. The order of the nodes is generally outputs, inputs/outputs, inputs, but each ROOMMS element type has its own requirements which can be seen later in this appendix.

model_name is a **model_name** defined in an **model card**.

No model parameters are permitted on element cards. Any extra information on an element card will be ignored.

C.3.5 Simulation Parameters

All simulation parameters must be placed following the key line that follows:

Simulation Parameters

Simulation parameter cards are of the form:

`.simulation_parameter_keyword parameter(s)`

where: `.simulation_parameter_keyword` is one of the keywords shown in the following table.

`parameter(s)` is one or more parameters as explained in the table.

keyword	parameter(s)	explanation	default
<code>.time</code>	<code>mrt, stoptime</code>	minimum resolvable time step of simulation, and the stop time of the simulation in units of seconds.	none
<code>.llevels</code>	<code>low_threshold, high_threshold</code>	implicit voltage-to-logic conversion levels ¹	1.5v and 3.5v
<code>.vlevels</code>	<code>logic_0_voltage, logic_1_voltage, logic_X_voltage</code>	implicit logic-to-voltage conversion levels ²	0.0, 5.0, and 2.5
<code>.nstrengths</code>	<code>number_of_logic_strengths</code>	defines number of logic driving strengths which are available to be used. More-or-less unlimited in size ³	3

¹. parameters used to convert from voltage values to logic values when an input of a logic gate is connected to a analog node. Voltage levels less than `low_threshold` are considered a logic 0, voltages greater than `high_threshold` are considered a logic 1, and voltages in between `low_threshold` and `high_threshold` are considered a logic X. The driving strength is assumed to be `number_of_logic_strengths - 1`.

². parameters used to convert from logic values to voltage values when an input of an analog element is connected to a logic node. A logic 0 is mapped to the `logic_0_voltage`, a logic 1 is mapped to the `logic_1_voltage`, and a logic X is mapped to the `logic_X_voltage`.

keyword	parameter(s)	explanation	default
.mindvsch	minimum_delta_V	minimum change in voltage required during analog simulation in order to propagate a change ⁴	1.0e-3
.abstol	absolute_tolerance	part of number used to determine whether an analog node has converged to a solution ⁵	1.0e-6
.reltol	relative_tolerance	part of number used to determine whether and analog node has converged to a solution ⁶	1.0e-3
.sor	successive_over-relaxation_parameter	experimental number which may speed up analog simulation ⁷	1.0
.startat	start_time	alternate start time of simulation in units of seconds	0
.glitches	glitch_flag	either 0 or 1, reports glitches as they occur during logic simulation in output file - not particularly useful	0 (off)

⁴ The driving ability of a logic gate during logic simulation is modelled by driving strength. This parameter sets the number of available logic strengths. The number must be any integer greater than or equal to 3 and two less than the maximum integer size of the computer (i.e. 65K).

⁵ this parameter controls the amount of "relaxation" that occurs during an analog simulation. Should be set to the maximum value tolerable to the user.

⁶ with relative tolerance multiplied by the signal voltage on the node, the absolute tolerance is the tolerance within which the iterations must fall in order to be successful. Can be increased to speed up the simulation at the cost of inaccuracy.

⁷ with absolute tolerance, relative tolerance multiplied by the voltage signal on the node is the tolerance within the iterations must fall in order to be successful. Can be increased to speed up the simulation at the cost of inaccuracy.

⁸ multiplied by the initial guess of a node's signal voltage, this parameter can sometimes reduce the number of iterations of nodes at a timestep.

C.3.6 Output Requests

The last cards in the ROOMMS source deck must be the output requests. This consists of the keywords:

Output Requests

followed by a list of node_name(s) on one or more of the following lines in the following format (The node_name(s) are assumed to have been defined in element cards):

<group_number_x> node_name_1 node_name_2

<group_number_y> node_name_3 node_name_4

The <group_number_x> identifies an output "group" for analog nodes. This exists to allow a graphical display program to separate and group analog outputs, and is optional.

Note that ROOMMS implements only one output format, and the output file is the one specified on the command line. This output format is rough, and is explained further later. Graphical processing of the output must be done by third-party software.

C.4 ROOMMS Logic Element Types

The following logic gates elements are defined within ROOMMS:

gate element name	description	minimum # of inputs
Inverter	inverter gate	1

gate element name	description	minimum # of inputs
Buffer	buffer gate	1
Delay	delay gate	1
Or	or gate	2
Nor	nor gate	2
And	and gate	2
Nand	nand gate	2
Xor	exclusive or gate	2
Xnor	not exclusive or gate	2
Dflipflop	edge-triggered, clocked, set-reset D flipflop with enable	2
Jkflipflop	edge-triggered, clocked set-reset JK flipflop with enable	1
Geaptr	grounded capacitor (defined as analog element) [*]	-

The model card format of each logic gate (except flipflops) is of the form:

.model model_name gate_element_name : element_parameters

where the element parameters are given by:

#	parameter name	definition	default, units
0	tr	rise time	0 seconds
1	tf	fall time	0 seconds
2	trc	capacitive drive capability of the gate for a 0 to 1 transition	0 volts/amps
3	tfc	capacitive drive capability of the gate for a 1 to 0 transition	0 volts/amps
4	ci	input capacitance (of all inputs)	0 F
5	co	output capacitance (of all outputs)	0 F

^{*} grounded capacitor is available to model delay due to capacitive loads on output of logic gate element.

#	parameter name	definition	default, units
6	0str	driving strength of gate when output is 0	number of logic strengths (default = 3)
7	1str	driving strength of gate when output is 1	number of logic strengths
8	Xstr	driving strength of gate when output is X	number of logic strengths

The format of the element cards of all the logic gate elements except the flipflops is similar to the general case seen in Section C.3.4:

`.element element_name output_node_name input_node_name(s) model_name`

Inverter, Buffer and Delay gate elements have a maximum of 1 input node. All the other gates, except the flipflops have 2 or more inputs (the maximum number is not defined).

C.4.1 Flipflops

The model card format of the flipflops is of the form:

`.model model_name flipflop_element_name : element parameters`

where the element parameters are given by:

#	parameter name	definition	default, units
0	tcq	propagation delay between rising edge of Clock and Q, and set or reset to Q, and recovery time between set or reset and clock, and recovery time between enable and output	0 seconds
1	tdcs	set-up time period for input and rising edge of Clock where input propagated	0 seconds
2	tpw	clock pulse width plus hold time before clock	0 seconds

#	parameter name	definition	default, units
3	trc	capacitive drive capability of the gate for a 0 to 1 transition	0 volts/amps
4	tlc	capacitive drive capability of the gate for a 1 to 0 transition	0 volts/amps
5	ci	input capacitance (of all inputs)	0 F
6	co	output capacitance (of all outputs)	0 F
7	0str	driving strength of gate when output is 0	number_of_logic_strengths (default = 3)
8	1str	driving strength of gate when output is 1	number_of_logic_strengths
9	Xstr	driving strength of gate when output is X	number_of_logic_strengths

The format of the element card for the D flipflop is as follows:

.element element_name Q Qbar D Clock Set Reset Enable model_name

Q or Qbar must exist. D and Clock must exist. All the others can exist or the position must contain the keyword "floating". The default values for floating nodes are:

#	output or input	default value if "floating"
0	Q	output - no default
1	Qbar	output - no default
2	D	mandatory input
3	Clock	mandatory input
4	Set	1
5	Reset	1
6	Enable	1

The format of the element card for the JK flipflop is as follows:

.element element_name Q Qbar J K Clock Set Reset Enable model_name

Q or Qbar must exist. Clock must exist. All the others can exist or the position must contain the keyword "floating". The default values for floating nodes are:

#	output or input	default value if "floating"
0	Q	output - no default
1	Qbar	output - no default
2	J	1
3	K	1
4	Clock	mandatory input
5	Set	1
6	Reset	1
7	Enable	1

C.5 Analog Element Types

The following analog elements are defined within ROOMMS:

analog element name	description	# of nodes
Nmos	nMOSFET	4
Pmos	pMOSFET	4
Resistor	resistor	2
Fcapr	floating	2
Gcapr	grounded capacitor. (defined as analog element)	1

The model card format of each analog element is of the form:

```
.model model_name analog_element_name : element_parameters
```

C.5.1 MOSFETs

Model parameters for the MOSFET models are given below:

Parameter Number	definition	default value and units
0	W = width	1 m
1	L = length	1 m
2	Area of Source	unused
3	Area of Drain	unused
4	γ = bulk threshold parameter	0 $\sqrt{v}$
5	ϕ = surface potential	0.6 v
6	λ = channel-length modulation	0 v^{-1}
7	V _{BB} = voltage on bulk	unused
8	CJ = zero-bias perimeter capacitance	unused
9	CSS = gate-source overlap capacitance	F/m
10	CGG = gate-bulk overlap capacitance	unused
11	CDD = gate-drain overlap capacitance	F/m
12	V _{T0} = zero-bias threshold voltage (V _{BS} = 0)	+ 1.0 v for n-channel - 1.0 v for p-channel
13	KP = transconductance parameter = $\mu C'_{ox}$	20e-06 v

The order of the parameters is necessary.

The element card format for the MOSFETs follows:

```
.element mos_element_name drain_node_name gate_node_name source_node_name
bulk_node_name Nmos
```

The bulk_node_name is optional. Note that the drain and source are bidirectional input/outputs.

C.5.2 Resistor, Fcapr Capacitor

The Resistor element model and the Fcapr capacitor element model require one model parameter each, R and C, respectively.

The Resistor and Fcapr element card require two node connections. These two nodes are bidirectional. The order is unimportant.

C.5.3 Grounded Capacitor, Gcapr

The Gcapr model card requires one parameter, C.

The Gcapr element requires one node connection. The other node of the Gcapr element is connected to ground internally.

For logic simulation, the Gcapr element adds to the capacitive load on a node, along with the input and output capacitances of the elements connected to the node, and this capacitance value is used in the delay model calculations described in Section 3.2.1.1.

For analog simulation, the Gcapr element adds a conductance value to the node. A value of capacitance on each node, which can represent parasitic capacitance, can aid in the stability of the integration algorithm used (See Section 2.7.2).

C.6 ROOMMS Source Element Types

The following source elements are defined within ROOMMS. These source elements are used as signal drivers of the circuits, as explicit logic to voltage and voltage to logic converters, and as behavioural elements. Note that the output node of a source element can be driven only by that source element and that the signal value on that node is determined only by the source element, not by the simulation algorithm (excepting current source).

source element name	description
Vcls	Voltage-controlled logic source
Lcvs	Logic-controlled voltage source
Vcvs	Voltage-controlled voltage source
Lcvcvs	Logic-controlled voltage-controlled voltage source
Lsrc	Logic source
Tsrc	Timing source
Sinusoid	Sinusoid timing source
General	General timing source
Lrail	Logic rail - constant source
Vrail	Voltage rail - constant source
Isourceelement	Constant current source

C.6.1 Voltage-Controlled Logic Source

This source is an explicit voltage to logic conversion element. It has the following parameters:

#	parameter name	definition	default, units
0	v0	low threshold	low_threshold from .levels
1	v1	high threshold	high_threshold from .levels
2	0str	driving strength of gate when output is 0	number_of_logic_strengths - 1 (default = 3)
3	1str	driving strength of gate when output is 1	number_of_logic_strengths - 1
4	Xstr	driving strength of gate when output is X	number_of_logic_strengths - 1

This source element card has the form:

.element source_element_name output_logic_node input_analog_node VcIs

C.6.2 Logic-Controlled Voltage Source

This source is an explicit logic to voltage conversion element. It has the following parameters:

#	parameter name	definition	default, units
0	tr	rise time	0 seconds

#	parameter name	definition	default, units
1	tf	fall time	0 seconds
2	v0	low voltage	logic_0_voltage from .vlevels
3	v1	high voltage	logic_1_voltage from .vlevels
4	vX	unknown voltage	logic_X_voltage from .vlevels

This source element card has the form:

.element source_element_name output_analog_node input_logic_node Lcvs

C.6.3 Voltage-Controlled Voltage Source

This source converts an input analog signal into another analog signal based on a delay and polynomial coefficients. It has the following parameters:

#	parameter name	definition	default, units
0	delay	delay	0 seconds
1	a0	coefficient	1
2	a1		0
3	a2		0
:	:		i

This element outputs the result of the following equation after a set delay:

$$V_{out} = a_0 + a_1 V_{in} + a_2 V_{in}^2 + a_3 V_{in}^3 + \dots$$

This source can be used as a sample-and-hold, using the delay parameter and the appropriate gain element. A Vcvs can be used as a sort of analog shift register if excited repeatedly before the specified delay.

This source element card has the form:

.element source_element_name output_analog_node input_analog_node Vcvs

C.6.4 Logic-Clocked Voltage-Controlled Voltage Source

This source converts an input analog signal into another analog signal based on a delay and polynomial coefficients. In addition, it only passes the input analog signal when the logic clock is high, and holds the input signal when the clock is low. It has the following parameters:

#	parameter name	definition	default, units
0	delay	delay	0 seconds
1	a0	coefficient	1
2	a1		0
3	a2		0
i	i		i

This element outputs the result of the following equation after a set delay when the logic clock is high:

$$V_{out} = a_0 + a_1 V_{in} + a_2 V_{in}^2 + a_3 V_{in}^3 + \dots$$

This source is a better sample-and-hold, using a low clock signal to hold the input analog signal. Otherwise, when the clock signal is high, the source acts like the Vcvs source.

This source element card has the form:

```
.element source_element_name output_analog_node input_analog_node logic_clock_node
```

```
Lcvcvs
```

C.6.5 Logic Source

This source allows the driving of a logic circuit with a periodic pulse waveform.

It has the following parameters:

#	parameter name	definition	default, units
0	Logic0	starting logic value	0
1	Logic1	second logic value	1
2	delay	time delay before pulse starts	0 seconds
3	period	period of pulse	- seconds
4	bp0	break point 0	- seconds
5	bp1	break point 1	- seconds
6	bp2	break point 2	- seconds
:	:		

The signal waveform is created as follows:

This source's element card has the form:

`.element source_element_name output_logic_node optional_node Lsrc`

The optional_node is ignored as it is assumed to be a ground node.

C.6.6 Timing Source

This source allows the driving of an analog circuit with a periodic pulse waveform. It has the following parameters:

#	parameter name	definition	default, units
0	V0	starting voltage value	0.0 volts
1	V1	second voltage value	5.0 volts
2	delay	time delay before pulse starts	0 seconds
3	period	period of pulse	- seconds
4	bp0	break point 0	- seconds
5	bp1	break point 1	- seconds

#	parameter name	definition	default, units
6	bp2	break point 2	- seconds
:	:		

The signal waveform is created as follows:

This source's element card has the form:

`.element source_element_name output_analog_node optional_node Tsrc`

The optional_node is ignored. It is assumed to be a ground node.

C.6.7 Sinusoid Timing Source

This source allows the driving of an analog circuit with a sinusoid waveform. It has the following parameters:

#	parameter name	definition	default, units
0	A	amplitude	0.0 volt/volt
1	Voffset	voltage offset (from 0.0 volts)	0.0 volts

#	parameter name	definition	default, units
2	delay	phase delay	0 seconds
3	period	period of pulse	- seconds

This source's element card has the form:

`.element source_element_name output_analog_node optional_node Sinusoid`

The optional_node is ignored. It is assumed to be a ground node.

C.6.8 General Timing Source

This source allows the driving of an analog circuit with a waveform generated from a polynomial. It has the following parameters:

#	parameter name	definition	default, units
0	a0	first coefficient	1.0
1	a1	second coefficient	0.0
2	a2	third coefficient	0.0
⋮	⋮		

The signal waveform is created as follows:

$$V_{out} = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + \dots$$

This source's element card has the form:

.element source element_name output_analog_node optional_node Gensrc

The optional node is ignored as it is assumed to be a ground node.

C.6.9 Logic Rail - Constant Source

This source allows the driving of a logic node with a constant logic value. It has the following parameters:

#	parameter name	definition	default, units
0	LogicValue	constant logic value	0

This source's element card has the form:

.element source element_name output_logic_node optional_node Lrail

The optional node is ignored as it is assumed to be a ground node.

C.6.10 Voltage Rail - Constant Source

This source allows the driving of an analog node with a constant voltage value.

It has the following parameters:

#	parameter name	definition	default, units
0	V0	constant voltage value	0.0 volts

This source's element card has the form:

.element source_element_name output_analog_node optional_node Vrail

The optional_node is ignored as it is assumed to be a ground node.

C.6.11 Constant Current Source

This source allows the addition of a constant current to an analog node. This source acts like any other analog element by adding a conductance and current value to an analog during the Iterated Timing Analysis algorithm. It does not drive a node like the rest of the sources. It has the following parameters:

#	parameter name	definition	default, units
0	G	conductance	0.0 mhos
1	I	current	0.0 amps
2	C	capacitance	0.0 F

This source's element card has the form:

.element source_element_name output_analog_node optional_node Isourceelement

The optional_node is ignored as it is assumed to be a ground node.

C.7 ROOMMS Output File Format

The ROOMMS output file format is an arbitrary format of the following form (keywords are in bold):

```
Simulation Title up to 80 characters length

TimeStepSize = float_value_in_seconds
NumberOfLogicStrengths = integer_value
Signals:
node_name_1:type("Analog:I/OFlag"_or_"Logic")
node_name_2:type("Analog:I/OFlag"_or_"Logic")
node_name_3:type("Analog:I/OFlag"_or_"Logic")
.
.
.
node_name_n:type("Analog:I/OFlag"_or_"Logic")
Results:
analog node_name_x:timestep(integer):signal
logic node_name_y:timestep(integer):signal_logic_value:signal_strength
analog node_name_z:timestep(integer):signal
.
.
.
```

where: "I/OFlag" is an integer, with a value corresponding to the output "group" of the analog node

"signal" is a float for an analog node (units of volts)

"signal_logic_value" is an integer (0 for logic low, 1 for logic high, and 2 for logic unknown)

"signal strength" is an integer

Note that the results come out after the keyword "Results" in order for each respective node, but that each node is printed in the order of evaluation. The result is that

the order of output does not follow sequential time. In addition, the output is not continuous, and only occurs when the signals changes. Each node's output is piecewise continuous.

All information required to translate the output of ROOMMS for a graphics display program is contained in this format. A filter program has been created to convert ROOMMS output into a third party graphics file format [38].

Appendix D An Example ROOMMS Input Deck

The following source deck contains the circuit description for the circuit of Figures 4-5 and 4-6, the analog part of an in-service switched-capacitor pulse shaping filter [6,40]. The process parameters are those of the Canadian Microelectronics Company [43].

ROOMMS41 filter circuit

Model Cards

```
.model n12tran nmos :40e-6 3e-6 220.0e-12 340.0e-12 1.1 0.6 0.01 -2.5 4.4e-4 0.0 0.0
0.0 0.7 50e-6
.model n13tran nmos :38.5e-6 3e-6 0.0 340.0e-12 1.1 0.6 0.01 -2.5 4.4e-4 0.0 0.0 0.0
0.7 50e-6
.model n22tran nmos :40e-6 3e-6 60.0e-12 340.0e-12 1.1 0.6 0.01 -2.5 4.4e-4 0.0 0.0
0.0 0.7 50e-6
.model n30tran nmos :250e-6 4e-6 0.0 440.0e-12 1.1 0.6 0.01 -2.5 4.4e-4 0.0 0.0 0.0
0.7 50e-6
.model n33tran nmos :70e-6 3e-6 0.0 440.0e-12 1.1 0.6 0.01 -2.5 4.4e-4 0.0 0.0 0.0
0.7 50e-6

.model p10tran pmos :160e-6 6e-6 880.0e-12 880.0e-12 0.6 0.6 0.03 2.5 1.5e-4 0.0 0.0
0.0 -0.8 16e-6
.model p11tran pmos :80e-6 3e-6 440.0e-12 680.0e-12 0.6 0.6 0.03 2.5 1.5e-4 0.0 0.0
0.0 -0.8 16e-6

.model c2node gcapr: 10.0e-12
.model cnode gcapr: 0.1e-12

.model vb1 vrail : 0.9501
.model vb2 vrail : 0.4933
.model vb3 vrail : -0.8837
.model vb4 vrail : -1.3108

.model nswitch nmos :6e-6 3e-6 45.0e-12 45.0e-12 1.1 0.6 0.01 -2.5 4.4e-4 0.0 0.0 0.0
0.7 50e-6
```

```
.model pswitch pmos :5e-6 3e-6 45.0e-12 45.0e-12 0.6 0.6 0.03 2.5 1.5e-4 0.0 0.0 0.0
-0.8 16e-6
```

```
.model n1srg nmos :6e-6 3e-6 45.0e-12 45.0e-12 1.1 0.6 0.01 -2.5 4.4e-4 0.0 0.0 0.0
0.7 50e-6
```

```
.model n2srg nmos :6e-6 3e-6 45.0e-12 45.0e-12 1.1 0.6 0.01 -2.5 4.4e-4 0.0 0.0 0.0
0.7 50e-6
```

```
.model cfmodel gcapr: 2e-12
.model c5model fcapr: 0.743e-12
.model c6model fcapr: 0.979e-12
.model c7model fcapr: 0.696e-12
.model c8model gcapr: 0.0402e-12
.model c9model fcapr: 0.884e-12
.model c10model fcapr: 1.914e-12
.model c11model fcapr: 2.927e-12
.model c12model fcapr: 1.855e-12
.model camodel fcapr: 0.362e-12
.model ccmmodel fcapr: 0.5e-12
.model clmodel gcapr: 2.0e-12
.model cbmodel fcapr: 0.1e-12
```

```
.model c2model gcapr: 0.01e-12
```

```
.model rfmodel tres: 100e3
```

```
.model rdcmodel tres: 1e9
```

```
.model vref vrail : -1.8
```

```
.model vdd vrail : 2.5
```

```
.model vss vrail : -2.5
```

```
.model gnd vrail : 0.0
```

```
.model vphi1 tsrc: -2.5, 2.5, 0, 390.625e-9, 0, 25e-9, 45e-9, 175e-9, 195e-9
```

```
.model vphi1b tsrc: 2.5, -2.5, 0, 390.625e-9, 0, 25e-9, 45e-9, 175e-9, 195e-9
```

```
.model vphi2 tsrc: -2.5, 2.5, 0, 390.625e-9, 0, 220e-9, 240e-9, 370e-9, 390e-9
```

```
.model vphi2b tsrc: 2.5, -2.5, 0, 390.625e-9, 0, 220e-9, 240e-9, 370e-9, 390e-9
```

```
.model r1model tsrc:
```

```
2.5,-2.5,100e-9,6.25e-6,0,391e-9,391e-9,611e-9,631e-9,761e-9,781e-9,1001e-9,1021e-9,
1151e-9,1171e-9,1392e-9,1412e-9,1542e-9,1562e-9,1783e-9,1803e-9,1933e-9,1953e-9,
2173e-9,2193e-9,2323e-9,2343e-9,2564e-9,2584e-9,3691e-9,3711e-9,3931e-9,3951e-9,
4081e-9,410,1e-9,4322e-9,4342e-9,4472e-9,4492e-9,4713e-9,4733e-9,4863e-9,4883e-9,
5103e-9,5123e-9,5253e-9,5273e-9,5494e-9,5511e-9,5644e-9,5664e-9,5859e-9,5859e-9
```

.model b1model tsrc:

-2.5,2.5,100e-9,6.25e-6,0,416e-9,436e-9,566e-9,586e-9,806e-9,826e-9,956e-9,976e-9,
1197e-9,1217e-9,1347e-9,1367e-9,1588e-9,1608e-9,1738e-9,1758e-9,1978e-9,1998e-9,
2128e-9,2148e-9,2369e-9,2389e-9,2519e-9,2539e-9,3736e-9,3756e-9,3886e-9,3906e-9,
4126e-9,4146e-9,4276e-9,4296e-9,4517e-9,4537e-9,4667e-9,4687e-9,4908e-9,4928e-9,
5058e-9,5078e-9,5298e-9,5318e-9,5448e-9,5468e-9,5689e-9,5709e-9,5839e-9,5859e-9

.model r2model tsrc:

2.5,-2.5,100e-9,6.25e-6,0,761e-9,781e-9,1001e-9,1021e-9,1151e-9,1171e-9,1392e-9,
1412e-9,1542e-9,1562e-9,1783e-9,1803e-9,1933e-9,1953e-9,2173e-9,2193e-9,4081e-9,
4101e-9,4322e-9,4342e-9,4472e-9,4492e-9,4713e-9,4733e-9,4863e-9,4883e-9,5103e-9,
5123e-9,5253e-9,5273e-9,5494e-9,5511e-9

.model b2model tsrc:

-2.5,2.5,100e-9,6.25e-6,0,806e-9,826e-9,956e-9,976e-9,1197e-9,1217e-9,1347e-9,
1367e-9,1588e-9,1608e-9,1738e-9,1758e-9,1978e-9,1998e-9,2128e-9,2148e-9,4126e-9,
4146e-9,4276e-9,4296e-9,4517e-9,4537e-9,4667e-9,4687e-9,4908e-9,4928e-9,5058e-9,
5078e-9,5298e-9,5318e-9,5448e-9,5468e-9

.model r3model tsrc:

2.5,-2.5,100e-9,6.25e-6,0,1151e-9,1171e-9,1392e-9,1412e-9,1542e-9,1562e-9,1783e-9,
1803e-9,4472e-9,4492e-9,4713e-9,4733e-9,4863e-9,4883e-9,5103e-9,5123e-9

.model b3model tsrc:

-2.5,2.5,100e-9,6.25e-6,0,1197e-9,1217e-9,1347e-9,1367e-9,1588e-9,1608e-9,1738e-9,
1758e-9,4517e-9,4537e-9,4667e-9,4687e-9,4908e-9,4928e-9,5058e-9,5078e-9

.model r8model tsrc:

-2.5,2.5,100e-9,6.25e-6,0,220e-9,240e-9,370e-9,390e-9,611e-9,631e-9,761e-9,781e-9,
1001e-9,1021e-9,1151e-9,1171e-9,1392e-9,1412e-9,1542e-9,1562e-9,1783e-9,1803e-9,
1933e-9,1953e-9,2173e-9,2193e-9,2323e-9,2343e-9,2564e-9,2584e-9,2714e-9,2734e-9,
2954e-9,2974e-9,3300e-9,3320e-9,3541e-9,3561e-9,3691e-9,3711e-9,3931e-9,3951e-9,
4081e-9,4101e-9,4322e-9,4342e-9,4472e-9,4492e-9,4713e-9,4733e-9,4863e-9,4883e-9,
5103e-9,5123e-9,5253e-9,5273e-9,5494e-9,5511e-9,5644e-9,5664e-9,5884e-9,5904e-9,
6034e-9,6054e-9

.model b8model tsrc:

2.5,-2.5,100e-9,6.25e-6,0,175e-9,195e-9,416e-9,436e-9,566e-9,586e-9,806e-9,826e-9,
956e-9,976e-9,1197e-9,1217e-9,1347e-9,1367e-9,1588e-9,1608e-9,1738e-9,1758e-9,
1978e-9,1998e-9,2128e-9,2148e-9,2369e-9,2389e-9,2519e-9,2539e-9,2759e-9,2779e-9,
2909e-9,2929e-9,3345e-9,3366e-9,3496e-9,3516e-9,3736e-9,3756e-9,3886e-9,3906e-9,
4126e-9,4146e-9,4276e-9,4296e-9,4517e-9,4537e-9,4667e-9,4687e-9,4908e-9,4928e-9,
5058e-9,5078e-9,5298e-9,5318e-9,5448e-9,5468e-9,5689e-9,5709e-9,5839e-9,5859e-9,
6079e-9,6099e-9

Element Cards

```
.element x24n an23 an31 an2 nswitch
```

```
.element x24p an23 an32 an2 pswitch
```

```
.element x25n an23 an33 gnd nswitch
```

```
.element x25p an23 an34 gnd pswitch
```

```
.element x26n an24 an31 an3 nswitch
```

```
.element x26p an24 an32 an3 pswitch
```

```
.element x28n an24 an33 gnd nswitch
```

```
.element x28p an24 an34 gnd pswitch
```

```
.element x5m1 an44 an64 an30 n1srg
```

```
.element x5m2 an44 an84 gnd n2srg
```

```
.element x6m1 an45 an65 an30 n1srg
```

```
.element x6m2 an45 an85 gnd n2srg
```

```
.element x7m1 an46 an66 an30 n1srg
```

```
.element x7m2 an46 an86 gnd n2srg
```

```
.element x9m1 an48 an68 an30 n1srg
```

```
.element x9m2 an48 an88 gnd n2srg
```

```
.element x10m1 an49 an69 an30 n1srg
```

```
.element x10m2 an49 an89 gnd n2srg
```

```
.element x11m1 an50 an70 an30 n1srg
```

```
.element x11m2 an50 an90 gnd n2srg
```

```
.element x12m1 an51 an71 an30 n1srg
```

```
.element x12m2 an51 an91 gnd n2srg
```

```
.element c5 an44 an20 c5model
```

```
.element c6 an45 an20 c6model
```

```
.element c7 an46 an20 c7model
```

```
.element c8 an47 c8model
```

```
.element c9 an48 an20 c9model
```

```
.element c10 an49 an20 c10model
```

```
.element c11 an50 an20 c11model
```

```
.element c12 an51 an20 c12model
```



```
.element ca an20 an23 camodel
.element cb an2 an3 cbmodel
.element cc an23 an24 ccmodel
.element cl an3 clmodel
.element cf an29 cfmodel
```

```
.element c2 an2 c2model
```

```
.element rdc an20 gnd rdcmodel
.element rf an3 an29 rfmodel
```

```
.element r1 an64 r1model
.element b1 an84 b1model
```

```
.element r2 an65 r2model
.element b2 an85 b2model
```

```
.element r3 an66 r3model
.element b3 an86 b3model
```

```
.element r5 an68 r3model
.element b5 an88 b3model
```

```
.element r6 an69 r2model
.element b6 an89 b2model
```

```
.element r7 an70 r1model
.element b7 an90 b1model
```

```
.element r8 an71 r8model
.element b8 an91 b8model
```

```
.element vdd an9 vdd
.element vss an10 vss
.element gnd gnd gnd
```

```
.element vphi1 an31 vphi1
.element vphi1b an32 vphi1b
.element vphi2 an33 vphi2
.element vphi2b an34 vphi2b
```

```
.element vref an30 vref
```

```
.element m10 an11 an5 an9 p10tran
```

```
.element m11 an4 an6 an11 p11tran
```

```
.element m12 an4 an4 an12 n12tran
```

```
.element m13 an12 an12 an10 n13tran
```

```
.element m20 an13 an5 an9 p10tran
```

```
.element m21 an3 an6 an13 p11tran
```

```
.element m22 an3 an7 an14 n22tran
```

```
.element m23 an14 an12 an10 n22tran
```

```
.element m30 an11 gnd an15 n30tran
```

```
.element m31 an13 an2 an15 n30tran
```

```
.element m33 an15 an8 an10 n33tran
```

```
.element vb1i an5 vb1
```

```
.element vb2i an6 vb2
```

```
.element vb3i an7 vb3
```

```
.element vb4i an8 vb4
```

```
.element cn14 an14 cnode
```

```
.element cn13 an13 cnode
```

```
.element cn11 an11 cnode
```

```
.element cn4 an4 c2node
```

```
.element cn12 an12 cnode
```

```
.element cn15 an15 cnode
```

Simulation Parameters

```
.time 2e-9 18.75e-6
```

```
.setnode an2 = -1.9171e-3 an3 = -3.8343e-3 an4 = 197.8505e-3
```

```
an5 = 950.1e-3 an6 = 493.3e-3 an7 = -883.7e-3 an8 = -1.3108
```

```
an9 = 2.5 an10 = -2.5 an11 = 1.9226 an12 = -1.4315 an13 = 1.9257
```

```
an14 = -2.1614 an15 = -0.9110304
```

Output Requests

```
<1> an2 an3 an29
```

```
<2> an64 an84 an20 an31
```

Appendix E Device Model Equations

Model equations are shown for analog elements, along with the model parameters in the order used by ROOMMS.

E.1 Resistor

a. Model Parameter

Parameter Number	definition	default value and units
0	R = resistance	0 Ω

b. equations:

$$G_n = \frac{1}{R}$$

$$I_n^s = \frac{((V_1^s - V_1^{s-k}) - (V_2^s - V_2^{s-k}))}{R}$$

E.2 Floating capacitor

a. Model Parameter

Parameter Number	definition	default value and units
0	C = capacitance	0 F

b. equations

$$G_{eq} = \frac{C}{h}$$

$$I_{eq}^n = \frac{C}{h} ((V_1^n - V_1^{n-1}) - (V_2^n - V_2^{n-1}))$$

E.3 MOSFET (SPICE level 1 model [9,23])

a. Model Parameters (from SPLICE1.7 [12]):

Parameter Number	definition	default value and units
0	W = width	1 m
1	L = length	1 m
2	Area of Source	unused
3	Area of Drain	unused
4	γ = bulk threshold parameter	0 $\sqrt{v}$
5	ϕ = surface potential	0.6 v
6	λ = channel-length modulation	0 v^{-1}
7	V_{BH} = voltage on bulk	unused
8	CJ = zero-bias perimeter capacitance	unused
9	CSS = gate-source overlap capacitance	F/m
10	CGG = gate-bulk overlap capacitance	unused
11	CDD = gate-drain overlap capacitance	F/m
12	V_{T0} = zero-bias threshold voltage ($V_{BS} = 0$)	+ 1.0 v for n-channel - 1.0 v for p-channel
13	KP = transconductance parameter = $\mu C'_{ox}$	20e-06 v

b. equations for triode region

$$\text{for } V_{GS} > V_{TH}, V_{DS} < V_{GS} - V_{TH}$$

$$I_{DS} = KP \frac{W}{L - 2X_p} \left(V_{GS} - V_{TH} - \frac{V_{DS}}{2} \right) V_{DS} (1 + \lambda V_{DS})$$

where:

X_{jl} = lateral diffusion from overlap of gate electrode over source and drain regions

(assumed to be $\ll L$)

$$V_{TH} = V_{T_0} + \gamma(\sqrt{2\phi_p} - V_{DS} - \sqrt{2\phi_p})$$

$$\phi_s = 2\phi_p = 2kT \ln \frac{N_A}{n_i} = \text{surface potential}$$

(1) Source node

$$G_s = \frac{KP}{2} \left(\frac{W}{L} - 2X_j \right) \left[(1 + \lambda V_{DS}) \left(V_{GS} - V_{TH} - \frac{1}{2} \frac{\gamma V_{DS}}{\sqrt{V_{SB} + 2\phi_p}} \right) + \lambda V_{DS} \left(V_{GS} - V_{TH} - \frac{V_{DS}}{2} \right) \right]$$

(2) Drain node

$$G_D = KP \frac{W}{L - 2X_j} \left[(1 + \lambda V_{DS}) (V_{GS} - V_{TH} - V_{DS}) + \lambda V_{DS} \left(V_{GS} - V_{TH} - \frac{V_{DS}}{2} \right) \right]$$

c. equations for **saturation** region

$$I_{EQ} = I_{DS} = \mu C_{ox} \frac{W}{2L} (1.0 + \lambda V_{DS}) (V_{GS} - V_{TH})^2$$

(1) Source Node

$$G_S = \mu C_{ox} \frac{W}{2L} \left[\lambda (V_{GS} - V_{TH})^2 + (1 + \lambda V_{DS}) (V_{GS} - V_{TH}) \left(2 - \frac{\gamma}{\sqrt{V_{SB} + 2\phi_p}} \right) \right]$$

(2) Drain node

$$G_D = \frac{\mu C_{ox}}{2} \frac{W}{L} [\lambda (V_{GS} - V_{TH})^2]$$

END

15-12-93

FIN