ABSTRACT

A new simulator was created to facilitate vertical circuit design integration. Written in Objective C, it readily allows the incorporation of new system/intermediate/device level models and modifications to the simulation algorithms. The analog simulation capability of the new simulator is demonstrated. A simulator model is presented and the new simulator's implementation of that model is shown. The necessary steps to add a new simulation capability, Elogic, are outlined. Code size and execution speed gains are discussed.

1.0 INTRODUCTION

It has been recognized that improved simulation tools are required to support the systematic design of increasingly complex integrated circuits [1,2,3,4,5]. The most useful tool would support vertically integrated design at the system level, through intermediate levels (including mixed analog/digital representations), down to the discrete device level. The development of such a tool requires the systematic use of a software environment which will ensure that the needs of the circuit designer are met, while supplying maximum flexibility to the simulator's designer(s).

The subject of this paper is ROOMMS, a Relaxation-based, Object-Oriented, Mixed-Mode Simulator. ROOMMS is a working, practical, mixed-mode simulator capable of intermediate and lower level simulations in the time domain (eg. for intermediate level analog circuits - controlled sources, for mixed analog and digital circuits, at the logic level - AND gates, and at the discrete device level - MOS transistors). ROOMMS has been written in the object-oriented programming language - Objective C [6]. The object-oriented concepts of classes, encapsulation, hierarchical organization and polymorphism (see Appendix A) speed up the programming process, and help the simulator's developer to create a complex program that is hierarchically structured, compartmentalized, easily modified, and understandable. The current ROOMMS implements the Iterated Timing Analysis [2,3] algorithm, and packages this proven analog algorithm with a proven relaxation-based logic algorithm [2]. In the following sections, the algorithms and their implementation compared to a simulator model are discussed. The design decisions required to add Elogic capabilities to ROOMMS are included. Output of ROOMMS is compared to existing simulators, and the development approach is discussed.

2.0 Summary of Simulator Algorithms

2.1 Iterated Timing Analysis

ROOMMS implements the non-linear relaxation-based analog simulation technique called Iterated Timing Analysis [2,3,4,7]. By only evaluating circuit nodes when they are active, Iterated Timing Analysis reduces the computation required compared to traditional matrix-based circuit simulation techniques [2,3]. A fixed timestep version was chosen for implementation. Simple analog elements, MOSFETs, resistors, and capacitors have been implemented, as well as more abstract controlled sources.

2.2 Logic Analysis

ROOMMS implements an event-driven, three-level (high, low and unknown) logic analysis technique. Logic gates such as ANDs, ORs and Inverters are available, as are higher level gates, such as JK Flip Flops. Rise and fall times can be specified, and a capacitive loading delay is implemented. Wired-ORs are possible using the concept of driving strengths of gates.

2.3 Elogic Analysis

Elogic is a simulation technique which uses analog element models and discrete signal voltage values [4,5]. 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. A further analysis of the addition of Elogic capabilities to ROOMMS follows.

3.0 ROOMMS

3.1 Implementation

ROOMMS has been designed in as modular a fashion as possible, in order to ease the development burden. Fig. 1 shows a conceptual model of the simulator. The main processes of the simulator, the Scheduler, Node, and Element, and the controlling Timestep and HandleTimeStep processes are complex in themselves, but have simple interfaces. The Scheduler process handles the event wheel, a record of which node is to be evaluated (or become an event) next, and also handles the scheduling of future events. A Node process determines the state of a signal on a node, either a voltage for analog simulations, or a logic state for logic simulations. Thus a Node process implements one of the main simulation algorithms. A Node process also determines whether a node should relax, or whether it should be scheduled again. An Element process is an implementation of the device model of a simulation element. The controlling HandleTimeStep and Timestep processes are simple do-until-done loops in the fixed timestep ROOMMS implementation. These few modules have been implemented directly within the simulator, using the inherent modularity of an object-oriented programming system.

Applying the simulator model of Fig. 1 to a real simulator involves mapping the functions shown and the data...
structures implied to a realizable form. The use of the object-oriented concepts of the class, a class hierarchy, encapsulation, and polymorphism in the development of ROOMMS has led to the development of a modular program, which follows this model very closely. Fig. 2 shows a class hierarchy of the ROOMMS simulator. The processes of Fig. 1 have mapped into the class structure quite elegantly. The Scheduler process, with the associated event queues, is implemented within the class Scheduler. A separate Node process is implemented for each simulation type (analog and logic) in the classes AnalogNodes and LogicNodes. Associated data, such as the signal value on the node are also defined within these data structures. The Element process of Fig. 1 has been implemented in a number of classes - one for each device model implemented for each simulation type. These classes also define the data structures associated with each device model (its device parameters). These classes include logic elements (AND, NAND, JK flip flops), analog elements (nMOSFET, resistors, capacitors) and sources (voltage-controlled voltage sources, piecewise linear sources, constant sources). The simpler control processes of Fig. 1 are defined within methods included in the Simulator class. Other classes in the class hierarchy of Fig. 2 consist of subclasses defining common data structures and behaviours, such as LogicElements, AnalogElements and Elements, and a few which define utilities and auxiliary data structures used during the simulation. For example, the class CircuitInput parses the source deck input to the simulator. The class LogicState contains a data structure which consists of the logic value (high, low or unknown), and the strength of the logic value (driving, weak or tri-state).

3.2 Addition of ELogic Analysis Capabilities

As an example of the flexibility of ROOMMS, the addition of a quasi-analog simulation capability, ELogic [4,5], was considered. Adding this capability requires implementing a new version of the Node and Element processes of the Fig. 1 model by creating a small number of new classes which define the new data structures and algorithm. These 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 differ in the way a signal value is determined and how an event is scheduled. Because of this close similarity, ELogicNodes would be implemented as a subclass of AnalogNodes. ELogicNodes would redefine only the method which determines the new voltage on a node and inherit all other behaviour and all data structure from AnalogNodes. 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. This would reduce 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 need 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. Thus only two main classes and the device models would be developed.

It is estimated that the addition of the ELogic capability requires the development of less than 400 lines of code (plus the device models). Very minor changes are required to the input and set-up routines of the existing simulator.

Figure 1  Simulator Model

Figure 2  ROOMMS Class Hierarchy

Figure 3  Boot-strapped Inverter
4.0 Results
ROOMMS capabilities have been verified by simulating a number of example circuits. A boot-strapped inverter [7] is presented here. Fig. 3 shows this circuit and the ROOMMS source deck, while Fig. 4 shows the ROOMMS output compared to HSPICE and SPLICE1.7 results. The output of the analog part of a mixed analog and digital transmit filter [8,9] is shown in Fig. 5, compared to HSPICE (HSPICE used timing sources to represent the behaviour, as verified by ROOMMS simulations, of the digital part of the filter). This is a pure analog simulation, consisting of a folded cascode opamp in an in-service switched capacitor pulse shaping filter. It includes 35 MOS transistors, 11 floating capacitors and an RC load. The output waveforms of Figures 4 and 5 clearly show the agreement between ROOMMS output and HSPICE.

![Figure 4 Boot-strapped Inverter Output](image_url)

![Figure 5 Transmit Filter Output](image_url)

5.0 Discussion
The preliminary development phase of ROOMMS, intended to show the capabilities of a simulator coded within the described environment, is coming to a close at the time of this paper. The environment has repeatedly shown that it can support the development of a capable simulator, by easing the coding of conceptual models, such as that shown in Fig. 1. (Appendix B sketches the development effort)

Throughput issues have not been addressed during this first phase except where they seriously affected results. Theoretically, an event-driven simulator based on the iterated Timing Analysis algorithm should be as accurate as a matrix-based simulator, and this is shown in the results. Some execution time saving would be anticipated due to circuit relaxation, and this execution time saving would be enhanced when higher level circuit models were used that were not available to matrix-based simulators like HSPICE. After the first phase of development, the ROOMMS execution speed performance has been slow. Inefficient coding is the problem, however. The slow speed is not necessarily a function of the development environment.

One of the prime reasons for using the object-oriented design environment with ROOMMS is that the simulator processes (Fig. 1) are implemented in a very modular fashion, with well-defined and strictly followed interfaces between modules. Therefore a module that is deemed to be inefficient in terms of execution time can be optimized without affecting the rest of the program. This will be the primary task of the next phase of the ROOMMS development. Large gains in execution time performance are expected.

A good performance comparison for this phase of development will be to compare ROOMMS' execution speed performance with a simulator, like SPLICE1.7, or iSPLICE3 [2,3,4,5,7], that implement similar algorithms. The object-oriented environment is not anticipated to degrade the performance in comparison with these two simulators, which are coded in Ratfor and "C". It is noted that if Objective C code cannot achieve the performance of these more conventional programming languages, Objective C code can be further optimized by using "C" code directly.

6.0 Conclusions and Acknowledgements
ROOMMS, a new simulator, and its design have been introduced. The design environment chosen, Objective C, has greatly sped up the program design process and eased the difficulty of adding enhancements to the basic simulator. The accuracy and ability of the simulator to simulate actual circuits has been demonstrated. Further work is required to enhance the throughput performance of the simulator.

The principal author would like to express his thanks to his close friend and co-author, Dr. T.A. Kwasniewski. He would also thank Rob Ivan for his contribution to the work. In addition, this work would not have been possible without the initial encouragement of Prof. M.A. Copeland, and the financial and time support of the principal author's employers at the Canadian Department of National Defence.

Appendix A Summary of Development Environment
A.1 Objective C and SMALLTALK
ROOMMS has been written in the object-oriented programming language - Objective C. Objective C is an implementation of the programming environment SMALLTALK [6,10]. Unlike SMALLTALK, Objective C is not an environment, but rather it 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. An object 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...
gate, \textit{ANDGate1}, for example. A class is the description of an object, \textit{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 \textit{AND} gate model peculiar to \textit{ANDGate1}. An object is told to do something by passing it a message, \textit{calculateNewOutput} for example. This message invokes one of the methods defined within the class \textit{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 \textit{AND} and \textit{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 \textit{AND} and \textit{OR} can be grouped into one superclass \textit{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 runtime system checks to see what the current class is of the object variable, and then calls up the method within that class's definition. By having a method in several class definitions, \textit{calculateNewOutput} in the classes \textit{AND} and \textit{OR} for example, a variable which may be of type \textit{AND} or \textit{OR} at run time can be sent the message \textit{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.

Appendix B Summary of Development Effort

B.1 Code Size and Development Resources

ROOMMS has been developed on a SUN network in the Electronics Department of Carleton University. It has required 2 man-years of effort, and consists of 10,000 lines of code (25% comments) contained in 52 Objective C classes.

References


5.3.4