NECTAR: A KNOWLEDGE-BASED FRAMEWORK FOR ANALOG CIRCUIT VERIFICATION

by
T. M. Kelessoglou

Memorandum No. UCB/ERL M90/112

4 December 1990

NECTAR: A KNOWLEDGE-BASED FRAMEWORK FOR ANALOG CIRCUIT VERIFICATION

by

T. M. Kelessoglou

Memorandum No. UCB/ERL M90/112

4 December 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

NECTAR: A KNOWLEDGE-BASED FRAMEWORK FOR ANALOG CIRCUIT VERIFICATION

by

T. M. Kelessoglou

Memorandum No. UCB/ERL M90/112

4 December 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

NECTAR: A Knowledge-Based Framework for Analog Circuit Verification

by

Theologos Michael Kelessoglou

NECTAR: A Knowledge-Based Framework for Analog Circuit Verification

by

Theologos M. Kelessoglou

Abstract

Present VLSI circuits involving the design of analog subcircuits have spurred the demand for more sophisticated analog CAD tools. This research addresses the problem of improving existing analog CAD tools, both individually and from the viewpoint of the general design process. NECTAR is a computer framework that implements the proposed improvement methods.

Electrical circuit simulators are the most used analog CAD tools. They are complicated programs characterized by convergence, user-interface, speed, and other limitations. Software redesign has resulted in numerous improvements but several harder problems remain. The first part of this research involves the examination of alternative methods for tool improvement. The proposed Expert-Emulation Approach is based on the observation that experienced users successfully complete simulation tasks with artful manipulations of input data and control options of the programs. Simulation expertise, acquired from experts, experimentation, and observation of novices, is incorporated in the knowledge base of NECTAR as rules. NECTAR rules identify patterns that lead to nonconvergence and apply appropriate corrective actions to the tool inputs.

As circuit designers have become dependent on a multitude of polymorphic CAD tools, the need for tool integration and automatic design management has emerged. CAD frameworks supply the software foundation to meet these needs. The second part of this research involves the improvement of analog CAD tools by their integration in a framework. The NECTAR framework unites a host of analog circuit verification tools, including several types of simulators, post-processors, editors, and the Expert Emulator. With the integra-

tion in NECTAR, tool interaction is automated, a shared database minimizes data flow and conversions, emphasis is moved from tools to design tasks, routine actions are automated, distributed computing environments can be accessed easily, and user errors and training requirements are reduced.

Particular emphasis is given to human-computer interaction issues. NECTAR's uniform, friendly user interface simplifies tools invocation and result presentation and takes advantage of modern computer hardware.

A final aspect of this research has been the choice of different programming models and languages for the implementation of the various concepts as computer programs.

Donald O. Pederson
Thesis Committee Chairman

To the memory of my father, *Μιχαήλ Θ. Κελέσογλου*

Acknowledgements

This dissertation would not have been possible without the support of several people. With the following acknowledgements I attempt to express my sincere gratitude to them.

I am indefinitely indebted to my advisor Professor Pederson for his indispensable guidance throughout this research and for his general help with the completion of my Ph.D. First, he gave me the opportunity to join the CAD Group at a time in my studies and life when the future seemed bleak. He suggested an interesting and challenging research topic and then provided advice and course corrections whenever I was off "painting the barn" or "pushing a string." He was always "there" to take care of academic problems and, perhaps more significantly, to offer his moral support during difficult moments of my personal and family life. He taught me writing and speaking skills by reading and correcting numerous drafts of reports and papers and by taking personal interest in my oral presentations. I thank him for showing me with his example that openness, honesty and frankness result in the most rewarding situations.

I thank Professors Brayton and Sarason for serving as the other two members of my thesis committee and for beneficial comments and discussions. I thank Professors Pederson, Newton, Sangiovanni-Vincentelli, and Brayton for establishing and leading a superb academic environment, the CAD Group at Berkeley. The financial support of the Semiconductor Research Corporation is gratefully acknowledged.

I thank Karti Mayaram for his contribution of ideas, for offering his insight on circuit design and simulation, for numerous most beneficial discussions, for his continuous encouragement, and for his companionship and friendship through the duration of this research.

I thank Roberto Guerrieri for the initial design of the LISP-based inference engine and for many helpful discussions. I thank Rick Spickelmier for his expert help with programming using LISP and the X Window System. Thanks are also due to Ron Gyurcsik for beneficial discussions in the early stages of this research.

I thank Brian Biehl of Tektronix, Bill McCalla and Dick Dowell of Hewlett-Packard, Takahide Inoue of Sony, and Al Ruehli of IBM, for valuable feedback on this research.

During my many years at Berkeley I had the fortune to interact with many students of the Electrical Engineering and Computer Sciences Department. I thank Nazli Gündes, Andreas Cangellaris, Amit Bhaya, Pantelis Tsoucas, Chris DeMarco, Yulin Liao, Tammy Huang, Bill Nye, Tim Salcudean, George Jacob, Güntekin Kabuli, Takis Konstantopoulos, Joe Higgins, Ted Baker, Giorgio Casinovi, Yannis Ioannidis, Peter Kennedy, Randy Cieslak, Karti Mayaram, Wayne Christopher, Ron Gyurcsik, Dave Burnett, Beorn Johnson, Jeff Burns, Res Saleh, Fabio Romeo, Tiziano Villa, Jack Lee, Nick Weiner, Linda Milor, Rick Spickelmier, Tom Quarles, Jyuo-Min Shyu, Young Kim, Andrea Casotto, Roberto Guerrieri, Don Webber, Brian Lee, Srini Devadas, Jaijeet Roychowdhury, Abhijit Ghosh, Bill Lin, Spyros Potamianos, Dave Harrison, Brian Okrafka, Gregg Whitcomb, Mary and Wendell Baker, Tom Laidig, Dave Gates, Cormac Conroy, Dev Chen, and many others whose names are located in unreadable blocks of my memory, for being wonderful people to work and to associate with.

Finally, I thank Brad Krebs, K.J. Pires, and Mike Kiernan, for the excellent administration and support of the CAD-Group computing facilities, and Shelley Sprandel, Beth Rhine, Irena Stanczyk-Ng, Gwyn Horn, Tom Boot, Mary Storelli, Beatrice Lamotte, Genevieve Thiebaut, and Sandy Camacho, for helping me with departmental administrative matters.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Circuit simulation, the task of modeling and numerically analyzing the performance of electrical circuits using computers, plays an essential role in the design of present-day integrated circuits (IC's). Electrical circuit simulation, the most widely used level of simulation, involves analytical models for the circuit elements that relate terminal voltages and currents. Circuit designers use simulation for two main purposes. During the initial phase of the design process, simulations are used to evaluate quickly design ideas and to compare alternative designs. In the later stages of the design, detailed (and possibly lengthy) electrical simulations are the main means of verifying, before an IC is fabricated, that the design specifications are met.

Designers use circuit simulation programs (simulators) routinely but with varying degrees of effectiveness [McCalla88]. Electrical simulators, such as SPICE [Nagel75], are complex, CPU-intensive computer programs characterized by convergence and other limitations. Efficient use of simulators depends on the designer's familiarity with the programs and knowledge of the underlying principles of simulation. Less experienced or infrequent users often encounter problems with simulators that offer little or no information on the cause of the problem or possible remedies. Such complications may lead to user frustration, waste of time and CPU resources, and eventually dislike and avoidance of the tools.

Numerous major and minor algorithmic and software enhancements have improved many of the convergence, user-interface, speed, and other properties of simulators. Nevertheless, some of the harder problems have defied solutions general enough to be in-

corporated in the programs. The first part of the research described in this dissertation involves the investigation of alternative methods of improving simulation programs. An improvement approach is proposed that has been driven by the observation that experienced users often are able to complete simulation tasks by carefully avoiding or overcoming program shortcomings. The implementation of the proposed method as a computer program, called Expert Emulator, has been investigated. The choices of a programming model, a knowledge representation scheme, and an implementation language, crucial in developing a useful tool, are described.

As circuit designers have become increasingly dependent on a host of computer-aided design (CAD) tools for the timely completion of new designs, so has the need for tool integration and automatic design management grown. CAD frameworks provide the software infrastructure to meet the integration and design management needs. CAD frameworks have been developed first for digital-circuit design. This has been a result of the higher degree of attention and importance reached by digital circuits compared to analog circuits, the proliferation of digital CAD tools, and because analog design could get by with few tools. Recently, however, ever increasing clock frequencies and the tendency toward integration of digital and analog subcircuits on the same chip have stimulated significant interest for analog CAD tools and frameworks.

Started as an attempt to generalize the program environment of the Expert Emulator, the research described in the second part of this dissertation involves the delineation and development of an analog CAD framework, called NECTAR. Since synthesis tools for all but a few types of analog circuits are still in experimental stage, analog design remains an iterative process, with manual intervention by the designer an essential part of each iteration. Electrical circuit simulators are the main CAD tools used by analog designers, including the designers of critical digital building blocks. Hence, simulators and other verification tools are of primary importance for NECTAR. This part of the research focuses on collecting an array of simulation-related tools in a single environment that uses a common database to minimize data flow and conversions, automates routine actions, emphasizes task completion, hides tool idiosyncrasies, and reduces user errors and training requirements. Particular emphasis is given to human-computer interaction issues, with the goal of providing a uniform, friendly interface that best takes advantage of modern computer

hardware.

The remaining chapters are organized as follows. Chapter 2 describes the investigation of alternative methods to overcoming limitations of simulation programs leading to the Expert-Emulation Approach. The choice of a programming model for the realization of the Expert Emulator is also presented in the same chapter. The organization of circuit-simulation expertise as rules, to be used by the Expert Emulator, is described in Chapter 3. Primitive representation objects are defined and rules are classified and illustrated with examples.

Chapter 4 presents the evolution of the analog-verification framework, NECTAR, and the various choices made in its design. Issues and decisions on the human-computer interface of the framework are described in Chapter 5. Versions of the interfaces for two different classes of hardware are illustrated. In addition, a special mechanism for the straightforward incorporation of new rules in the Expert Emulator is presented.

Chapter 6 describes the implementation of the Expert Emulator and NECTAR. The experiences with the programming platforms used (two each for the Expert Emulator and NECTAR) are presented. Finally, the main research conclusions are summarized in Chapter 7.

Chapter 2

Improving the Use of Simulation Programs

2.1 Overview

In this chapter, a new approach to the use of simulation programs is presented. In addition, the choice of a programming model for the implementation of the approach is described. The various capabilities of the new approach are classified and illustrated with examples in Chapter 3. The actual implementation is presented in Chapter 6.

Users of electrical circuit simulators, such as SPICE, often experience convergence and other problems with the simulators. Although program improvements have alleviated many shortcomings, some of the harder-to-solve problems remain. A novel approach to overcome such problems is proposed in this chapter. Based on observations on the actions of experienced users, the new approach, called the Expert-Emulator Approach, does not improve directly the simulation programs but instead it enhances the use of the programs. This is accomplished with appropriate modifications to the simulation input.

The software realization of the Expert-Emulator Approach is a program called the Expert Emulator that, based on incorporated expert knowledge, is able to identify problem patterns and suggest solutions. Both the Expert Emulator and SPICE are embedded in a controlling environment, called NECTAR, that aids novice and more experienced users by automatically acting like a human expert.

A software implementation of the Expert Emulator has been chosen based on the irregularity and ill-defined boundaries of the domain of simulation problems. Expert knowledge is represented and applied according to the production-system programming model. The program consists of an unordered collection of basic units, called rules. Rules correspond to units of human problem-solving knowledge. The inference engine, the production-system executer, runs the rules in an order related to the problem being solved. Rules can be added to the program incrementally as they are acquired, owing to the separation of domain knowledge and program control in production systems.

2.2 The Difficulty of Improving SPICE

The initial motivation for the research reported in this dissertation was provided by the presence of convergence and other limitations in the simulation program SPICE despite continuous improvements.

First released in 1972, SPICE has enjoyed widespread acceptance and use among companies and universities around the world. The success of SPICE has been attributed to the following:

- Electrical circuit simulation programs are complex software systems that incorporate many different algorithms. SPICE combines a "best set" of algorithmic procedures resulting in a "well-conditioned" package [Pederson84].
- Most types of circuit analysis, including nonlinear DC, nonlinear transient, AC, Fourier, pole-zero (in SPICE3), small-signal DC, distortion, noise, and sensitivity, are allowed.
- Circuits containing a wide range of nonlinear active circuit devices can be simulated.
- SPICE runs on most types of machines — from personal computers to workstations, minicomputers, mainframes, and supercomputers — under several operating systems.
- The source code of the program and executables are available in the public domain.

The advent of new circuit simulators with more cost-effective algorithms, such as RELAX [White86] and SPLICE [Saleh87], has not changed the preference and confidence that circuit designers show in SPICE [Cande86]. The above-mentioned simulators are not yet in wide

use, partly because they provide a limited number of analyses and can simulate only certain classes of circuits.

SPICE has gone through numerous major and minor versions. A review of the evolution of SPICE is given in Appendix A. Changes introduced in newer versions resulted in new simulation capabilities, an augmented applicability range, speed-ups, the elimination of implementation errors, and other improvements. Despite all the changes and enhancements, some problems with SPICE, most notably algorithmic convergence problems, persevere. Attempts at finding general solutions to those problems have reached several impasse points that show the inherent difficulty of the problems [Hailey87, Colon89, Quarles89].

The difficulty of overcoming persistent problems with SPICE prompted the following objective at the onset of this research:

Research Objective 1 *Investigation of alternative methods to overcoming* SPICE *problems. In particular, emphasis should be given to the implementation of such methods, so that they result in useful* CAD *tool(s).*

2.3 Observing the Expert User

This section describes several practical observations on experienced users that led to the novel method for improving the use of SPICE proposed in Section 2.4.

As mentioned previously, SPICE has become an indispensable tool for many circuit designers. Constantly having to deal with the idiosyncrasies of the program, designers have become *expert users* of SPICE. Their expertise becomes evident when they encounter problems with or limitations of the program. By nature, such problems do not always yield to rigorous analysis.

SPICE users do not just give up when problems arise. Novice and experienced users alike typically change the simulation input data and attempt a new simulation run. If the simulation fails again, they attempt a new modification and run, and so on until the problem is overcome or, as is often the case with novices, until expert help is sought. It may take many iterations before this process is over, especially for a novice, who, lacking the special knowledge about the simulation program, usually attempts semi-random mod-

ifications. On the other hand, the process for an experienced user is better structured and shorter. Experts rely on their experience to solve the problem. Figure 2.1 illustrates the steps that an expert takes when encountered with problems.

Step 1—Problem Classification: Experts have acquired a "sense" for recognizing the troublesome elements of a problematic case. They extract the essential information from

- the circuit at hand, and
- the simulation results including error messages,

and are able to recognize the type of the problem. Examples of classes of problems are wrong connections, positive feedback loops, and isolated circuit nodes.

Step 2—Corrective-Technique Application: Having categorized the problem, experts are able to use certain *corrective techniques*, that have been applied to similar problems in the past with success. There is usually an explanation for using such techniques. However, usually there is no rigorous proof that guarantees the results. For most cases, this limitation stems from the absence of a convergence theorem for SPICE, as mentioned in Section 3.4. Hence, these techniques are, in general, empirical "rules of thumb." Examples of such techniques are topological modifications, use of simulation control parameters, and introduction of parasitics.

Detailed examples of the application of both steps are presented in Chapter 3.

The important point to observe is that, using their experience and following the steps outlined above, designers are often able to overcome many of the problems with SPICE. They successfully use the program that has been made available to them to complete the task of accurately simulating their designs. This practice by the designers suggests an alternative approach to overcoming limitations of SPICE.

2.4 The Expert-Emulation Approach—NECTAR

To investigate alternative methods to improve the use of SPICE, a SPICE simulation is first analyzed from a systems' standpoint. Figure 2.2a shows the three basic elements in a simulation: the input to the simulator, the simulation program (SPICE), and the output

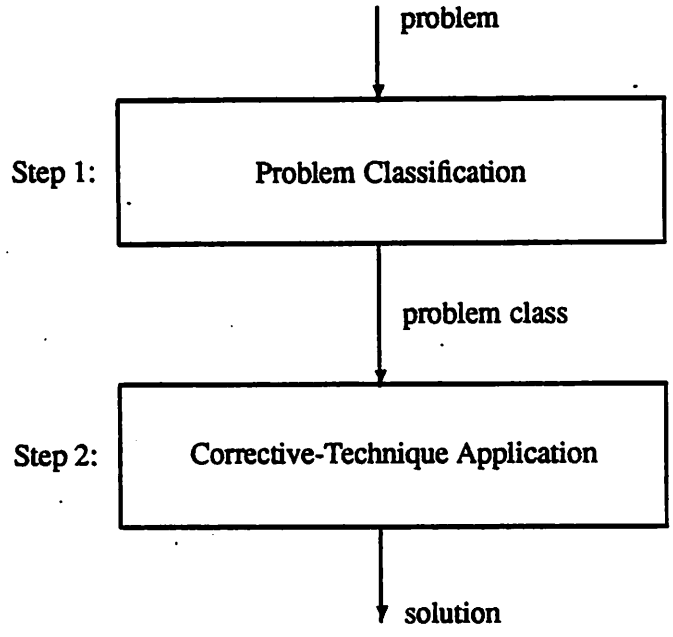


Figure 2.1: The expert's steps

generated during the simulation run. Assume that the simulation output is unsatisfactory because of shortcomings of the program. Then, a simple inspection of Figure 2.2 verifies the following:

Fact 1 (Simulation Improvement) *Consider a simulation program and an input to that program resulting in a certain (unsatisfactory) output. If the output can be improved, then the improvement will occur if and only if one of the following options occurs:*

O–1 The simulation program is improved.

O–2 The input to the simulator is modified appropriately.

O–3 Both of the above.

The first option, illustrated in Figure 2.2b, represents the natural, direct method to improve the simulation output, namely, code improvements or redesign. One can experiment with and modify subroutines in SPICE, fine tune device models, which are hard-coded in the program, improve the input compiler, so that it becomes able to recognize more errors, and fix implementation errors that exist in SPICE.

Given the size of the SPICE code and the complexity of the data structures and the overall flow of the program, understanding and modifying the code is not an easy task.

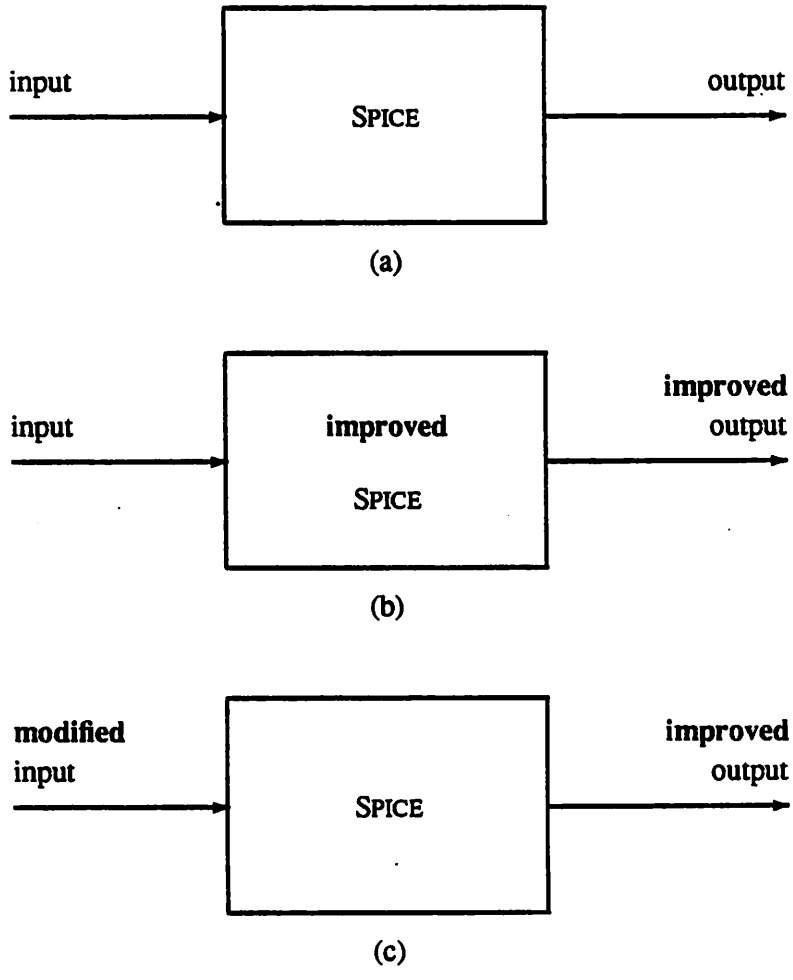


Figure 2.2: Illustration of the Simulation-Improvement Fact: (a) the basic elements of a simulation with SPICE; (b) the direct improvement method; (c) the alternative method

Even though individual algorithms in SPICE may be simple, their effect on the overall performance of the program is often complicated. Hence, any proposed change needs to be tested not just locally but in the context of the whole simulation package. These tests must involve a wide selection of benchmarks to ensure the global improvement. Similar remarks as made for SPICE's algorithms apply to the (hard-coded) device models. Electronic companies, such as AT&T Bell Labs, Harris, Hewlett Packard, SGS-Thompson, Tektronix, Texas Instruments, etc., and CAD companies, such as Daisy, Meta Software, MicroSim, NCSS, etc., have used this approach to develop their own proprietary versions of SPICE. These proprietary versions, known as "alphabet SPICE's" (ADVICE, DSPICE, HSPICE, HPSPICE, ISPICE, ISSPICE, PSPICE, TSPICE, TISPICE, etc.), have improved convergence and user-interface properties compared to the standard versions SPICE2G.6 and SPICE3c2.

The second option above, O–2, illustrated in Figure 2.2c, represents an alternative, indirect method. Since improving the program directly is a complex and difficult undertaking, this method attempts to sidestep problems employing appropriate modifications to the simulation input forcing the simulator to produce acceptable results. Clearly, the simulation problem described by the modified input must be the same with, or at least "close enough" to, the problem described by the original input. Otherwise, the new, improved simulation output would be worthless, as it would be the solution to the wrong problem.

Option O–2 makes an important implicit assumption, namely, that an appropriately modified input for the simulation is available. In reality, only the original input, that results in unsatisfactory output, is available at first. An improvement method based on Option O–2 should detail how the appropriately modified input is obtained.

Proposed here is the *Expert-Emulation Approach*, an improvement method based on Option O–2 and using the program *Expert Emulator* for the derivation of the modified input (Figure 2.3). As its name suggests, the Expert Emulator is a program emulating the actions of experienced SPICE users: it automatically performs the two steps of an expert, as outlined in Section 2.3, i.e., problem classification and corrective-technique application. The Expert Emulator, similar to a human expert user, takes its input from the original simulation input and from simulation results fed back from SPICE. The decisions of the Expert Emulator are based on domain knowledge collected from human experts and stored in the program in an appropriate form. The expert knowledge is made available to aid both

novices and more experienced users.

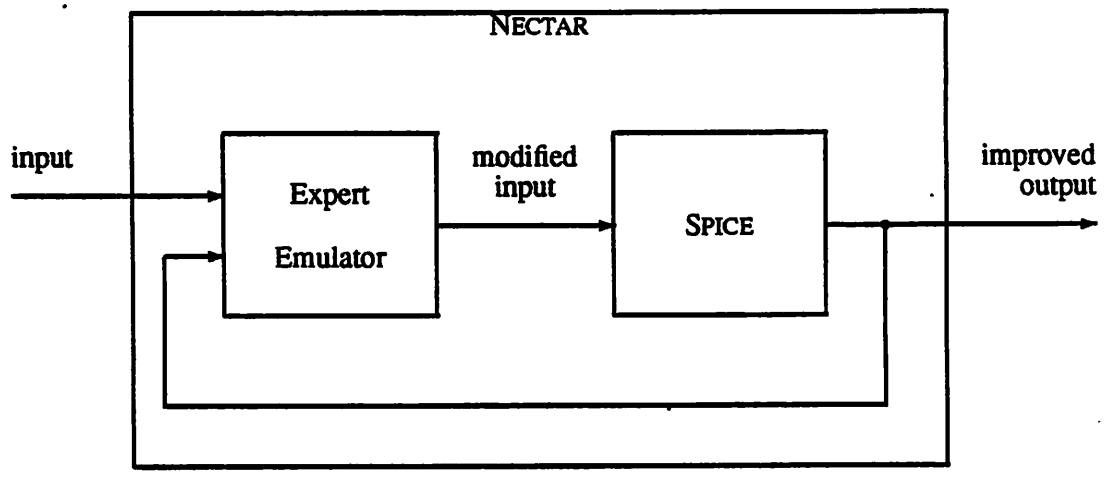


Figure 2.3: The Expert-Emulation Approach

The realization of the new technique as a computer program is described in Section 2.5. The computer environment that combines the Expert Emulator and SPICE, shown in Figure 2.3, is called *NECTAR*, an acronym for "kNowledge Environment for CAD Tools in the Analog Realm." In Chapter 4, NECTAR is extended to a more general environment combining several analog-circuit verification capabilities.

The purpose of the Expert-Emulation Approach is not to replace the direct method of Figure 2.2b, but to act as a parallel, complementary remedy. When a direct-approach improvement is made, it should supersede the corresponding Expert-Emulation-Approach solution. However, owing to the difficulty of many of the problems, the value of the Expert-Emulation Approach can be significant and lasting.

Occasionally, supposed improvements in the code of SPICE have resulted in worse results. This fact may be an explanation to what is generally the case, that designers tend to distrust new versions and stay with the older version they are accustomed to. In the Expert-Emulation Approach, SPICE is used unaltered. In this way, all the good properties that have made SPICE popular are retained and the new scheme is guaranteed to behave at least as well as SPICE on its own.

To use an analogy from systems, SPICE is treated as a "black box." The state of the simulator is observed only from its output, i.e., no additional internal probes are

set. Modifying the input is the only means of controlling the outcome of the simulation. Clearly, both the observability and the controllability of SPICE, a large and complex system, are limited. It follows that the possibilities that a given problem can be overcome using the Expert-Emulation Approach are also limited. The state of SPICE is simply of a much higher dimension than either its input or its output. Nevertheless, problems in the controllable subspace have the potential to be solved using the new approach.

The third option of Fact 1, O–3, involves both program and input modifications. If one assumes that the program changes are permanent, i.e., resulted in a new improved version, then O–3 degenerates into two simple successive steps, an O–1-type step followed by an O–2-type step. In essence, once the first step is completed, the improved SPICE becomes the new default version of the program. It is conceivable, however, that an older version might be better than the default version for a particular application. A user knowing this fact may want to use the older version when the occasion arises. The implication of the previous remark on the design of NECTAR is presented in Chapter 4.

2.5 Realization of the Expert Emulator

A software realization of the Expert-Emulation Approach should automate the two steps of an expert, as outlined in Section 2.3. The problem-classification step requires searching the space of incorporated types of simulation problems for patterns matching the problem at hand. The corrective-technique-application step involves the execution of certain procedures associated with the type of problem determined in the first step.

For reasons explained in Section 2.5.2, the realization of the Expert Emulator has been based on the *production-system* model of computation. This model is described in the next section.

2.5.1 Production Systems

Artificial Intelligence (AI) is that part of computer science that investigates reasoning processes, data representations, and other aspects of information systems that are able to perform tasks that would be thought to require intelligence if done by humans. Un-

reasonably high expectations and promises by the first AI researchers brought the field into disrepute among potential applicants. In recent years, however, the setting of more realistic, less general goals by the AI community has resulted in several successful scientific and engineering applications and has contributed to a big change in the general perception and recognition of the field [Barr82, Walker86]. Results of AI research, such as frames, semantic networks, rule-based systems, and others, are now seen not as panaceas but as useful programming tools.

Expert systems have been successful products of AI research. These systems, which vary widely in structure and behavior, all focus on methods of transferring knowledge from human experts to computer programs. The remainder of this section describes production systems, which are a type of expert systems that appear suitable for the realization of the Expert Emulator.

The familiar procedural programming model uses sequenced instructions as the basic unit of computation. In contrast, a production system uses a data-sensitive unordered collection of basic programming units, called *production rules* or simply *rules*.

Each rule has two parts. The *condition* part, or "if" part, or *Left-Hand Side* (LHS) of the rule describes the data configurations (patterns) for which the rule is appropriate. The *action* part, or "then" part, or *Right-Hand Side* (RHS) of the rule contains instructions for modifying the problem data when the rule is executed. The following is an example of a NECTAR rule in LISP-like pseudo-code.

```
(rule source-stepping
  (if
    (error (type convergence))
    (analysis (type dc)))
  (then
    (modify (source (value (ramp 0 dc))))
    (make (analysis (type transient)))))
```

This rule for SPICE use, named "source-stepping," looks for a convergence error during a DC analysis — in the LHS. The RHS instructs the conversion of DC sources to ramp sources (starting at 0 and ending at the DC value), followed by a request for a (pseudo-) transient analysis.

Both production systems and conventional procedural models have three major

components: the program, which expresses the computation to be performed and has the form of unordered rules and sequenced instructions, respectively; the executer, which performs the computation; and the data, which describes the problem to be solved and stores results. The executer in the procedural model simply executes the instructions in the order they are given in the program, unless specifically directed out of sequence by an instruction.

The executer in the production-system model, called the *inference engine*, is needed for a more complicated task; it must determine which rules are relevant to the current problem data and choose a rule to execute. The inference engine is a finite-state machine with a cycle consisting of three main states:

Match State: The inference engine checks the rule base against the problem data and finds all the rules with LHS's satisfied by the current data. The outcome of this state is called the *conflict set*. A single rule may be present in the conflict set multiple times, if its LHS is matched by different sets of objects in the data base.

Select State: During this state of the machine, the conflict set is ordered according to some selection strategy. These strategies typically use heuristics, such as the following:

- refraction: requiring that a rule can be executed at most once on the same data;
- data ordering: giving preference to rules that match data most recently added or accessed;
- specificity: favoring rules that are more specific according to some measure, such as the number of patterns in the LHS;
- rule ordering: statically and independent of the data;
- arbitrary or parallel selection: when everything else fails.

Execute State: One or more rules selected in the second state are passed to the third state for execution.

Since the rules usually change the data, the conflict set changes after each match-select-execute cycle. The inference engine halts when the conflict set becomes empty. Figure 2.4 illustrates the flow of control and data in a production system.

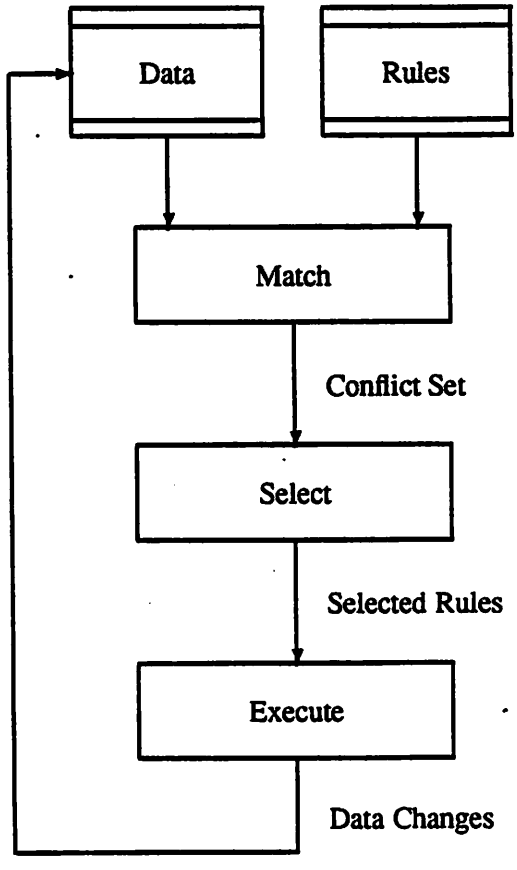


Figure 2.4: The flow of data and control in a production system

2.5.2 Choosing a Programming Model

This section outlines the considerations that led to a choice of programming model for the Expert Emulator.

The size of the problem space to be searched during the problem-classification step constitutes an important issue in designing the Expert Emulator. That size clearly depends on the amount of expert knowledge incorporated in the system. A "reasonably complete" Expert Emulator should contain expertise about many types of circuits, analyses, and simulation results. The problem space is a subset of the cross product of the spaces of circuit types, analysis types, and simulation-result types. Given the irregularity and huge size of the space of analog circuits alone, it follows that the search space can be huge.

Large search spaces tend also to be complicated. However, for some large problems, such as the traveling-salesman problem, there are efficient approximate algorithmic solutions. In general, algorithmic solutions exist for well-structured problems, i.e., problems having a rigid data format for which similar actions are performed for all data. When, on the contrary, there are many independent variables in the domain and responses must be diverse and based on attention to many factors, then a production system is an appropriate model. A procedural program would require a complex control structure to handle the switching to the appropriate code. The problem space for the Expert Emulator is not well structured.

In addition, the boundaries of the problem space can not be defined exactly and may be changing. This is another argument for the production-system choice, as such systems have the property of being able to cope with unanticipated situations. Unplanned but useful interactions result from applying knowledge (rules) when it is appropriate rather than calling on it in a predetermined sequence.

This last property is a consequence of what is considered the main advantage of production systems, the separation of domain expertise, contained in the rules, from the flow of control of the program, administered by the inference engine.

Because knowledge is stored in separate, nearly independent units, rules can be added to a production system with few side effects. This facilitates the addition of new rules to the knowledge base incrementally, as the rules are acquired.

The disadvantages of production systems compared to procedural programs are: slower speed of execution, because hardware is generally designed for procedural programs and, hence, an additional level of compilation, from rules to procedures, is needed; larger program size, since the inference engine must be included for the program to be able to execute; undesirable interaction among rules, which may result from the non-transparent behavior of the program, especially when the number of rules becomes large.

AI techniques have been used previously in CAD tasks, in particular when considerable amount of symbolic computation (alone or mixed with numerical computation) is required. In the area of circuit verification, the programs RUBICC [Lob84], CRITTER [Kelly84], DIALOG [DeMan85], QCritic [Bergquist86], and Critic [Spickelmier89] have used knowledge-based models. Knowledge-based synthesis systems include DAA [Kowalski85] and BLADES [ElTurky89].

After considering the various arguments, the choice of a production-system model for the Expert Emulator was made (illustrated in Figure 2.5). Further implementation considerations, including the choice of a programming language, are outlined in Chapter 6. The organization of the domain expertise in rules is described in Chapter 3.

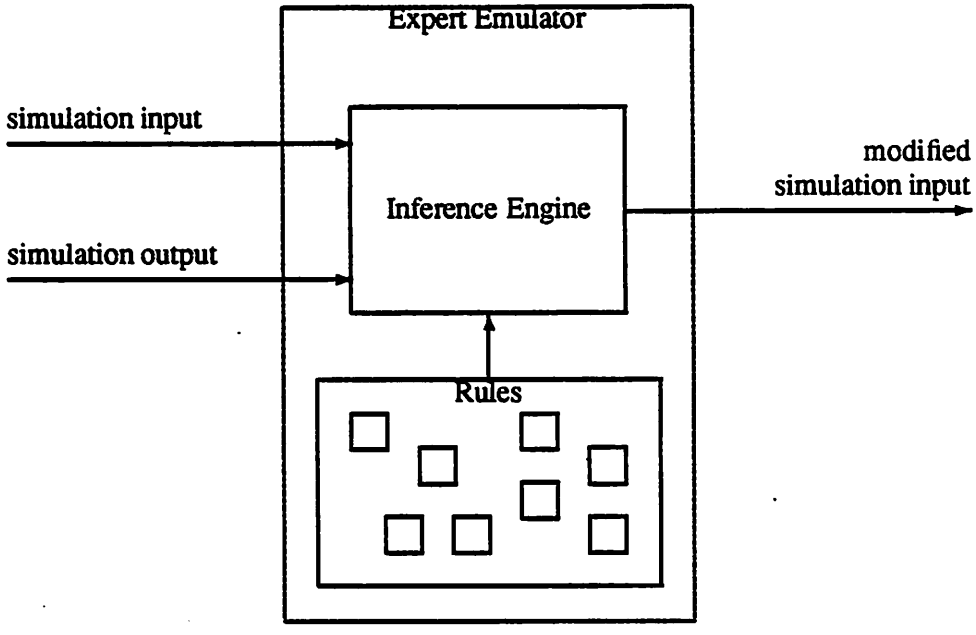


Figure 2.5: The Expert Emulator as a Production System

Chapter 3

Organizing Circuit Simulation Expertise as Rules

3.1 Overview

In this chapter, simulation expertise, which is the basis of the Expert-Emulation Approach, is organized as rules. The various capabilities of the Expert Emulator are classified and illustrated with examples.

A detailed examination of the space of simulation problems reveals the categories of data objects in the domain. A set of basic representation classes for the production-system model is declared to correspond with the objects of the domain. Expert-Emulator rules use the representation classes both for pattern matching during problem classification and for problem-data modifications during the application of corrective techniques.

Rules are divided in sets so that the performance of the production system does not degrade — a common problem of rule-based systems with many rules. The initial classification is made in three sets: simulation-error-recovery rules, presimulation rules, and design-aid rules.

For each rule set, the capabilities of the Expert Emulator are illustrated with examples of representative rules applied on test circuits.

3.2 Data-Representation Classes for the Expert Emulator

Expert knowledge must be incorporated in the Expert Emulator using a representation, such that intelligible inferences can be drawn. As explained in Section 2.5.2, the polymorphism of the problem data has been a deciding factor for the choice of a production-system model. Production systems generally are not strongly typed programming models. Data-structure declarations, when required at all, serve to specify the field names for compound data types but not the fundamental types (integer, floating-point number, character, etc.) of the individual fields. The latter are determined at run time. In this section, the basic classes of data for the production rules are defined, following a detailed examination of the problem space.

Data in the Expert Emulator can be divided in two main categories — input/output (I/O) data, used for the communication with the environment, and strictly internal data, used for storing intermediate results. As shown in Figure 2.5, the input to the Expert Emulator consists of the simulation input data and simulation output data, and output from the Expert Emulator consists of simulation input data. Hence, the set of I/O data for the Expert Emulator is the same as the set of I/O data for the simulator. Table 3.1 summarizes the categories of data in the domain. The simulation input consists of the circuit topology (net-

Expert-Emulator Data		
I/O Data	Simulation Input	circuit topology circuit-element values device models analysis requests simulator-control options
	Simulation Output	output-variable values error messages
Internal Data	abstractions of above	

Table 3.1: Data categories in the domain of the Expert-Emulator

list), values for circuit elements and parameters, model descriptions for nonlinear devices, circuit-analysis requests, and options to control the simulator. The simulator output consists

of arithmetic values for output circuit variables — voltages, currents, power consumption, poles and zeros, harmonics, distortion, sensitivities, etc. — and, when problems occur, error messages. Circuit variables can be scalars or vectors.

The apparent polymorphism of the I/O data is further compounded by the existence of many different types of circuit elements and by abstractions of the basic circuit elements via subcircuit definitions. The levels of abstraction for strictly internal data may vary considerably to correspond to expert knowledge of varying generality.

Table 3.2 lists the basic data classes for the production system. The basic simulation-input classes have been chosen to conform to the input language of SPICE. The latter is a compact and expressive language and has been adopted (sometimes with modifications) as the input language for several other simulation programs as well. The choice of basic classes similar to the basic types of the simulator minimizes data conversions. In addition to the simulation-input classes, two simulation-output classes are declared. They accommodate values for output variables and error messages, respectively. It should be noted that class fields are meant to store different fundamental types of data depending on the particular problem data. As an example, the "value" field of the "capacitor" class might contain any of the following data: 1, 200p, 4.7×10^{-3}, or (poly, 0.03, 10^{-6}). The "elements" field of the "subcircuit" class contains an arbitrary list of instances of circuit-element classes. Some fields, such as "turns-ratio" of "mutual inductance", are not specified in the SPICE input but are computed from other quantities.

Declarations for new classes can be added to the production system when the need arises. This capability is essential for the Expert Emulator, because it is impossible to derive a complete list of necessary classes in advance. In particular, hierarchical classes would be advantageous to the pattern-matching process, provided that the implementation language has an inheritance mechanism. Two different implementations of the Expert Emulator are described in Chapter 6.

Class Name	Fields
resistor	name term1 term2 value tc tc2 terminals
capacitor	name term1 term2 value ic terminals
inductor	name term1 term2 value ic terminals
mutual inductance	name inductor1 inductor2 coupling-coefficient turns-ratio
transmission line	name node1 node2 node3 node4 z0 td f nl ic
vccs	name term1 term2 control-term1 control-term2 value
vcvs	name term1 term2 control-term1 control-term2 value
cccs	name term1 term2 control-source value
ccvs	name term1 term2 control-source value
voltage source	name term1 term2 value
current source	name term1 term2 value
diode	name term1 term2 model area off ic
bjt	name collector base emitter bulk model area off ic
jfet	name drain gate source model area off ic
mosfet	name drain gate source bulk model l w ad as pd ps nrd nrs off ic
subcircuit	name nodes elements
model	name type af beta bf br bv cbd cbs cgd cgs cgbo cgdo cgso cj cjc cje cjo cjs cjsw delta eg eta fc gamma ibv ikf ikr irb is isc ise itf js kappa kf kp lambda ld level m mj mjc mje mjs mjsw n nc ne neff nf nfs nr nss nsub pb phi ptf rb rbm rc rd re rs rsh tf theta tox tpg tr tt ucrit uexp uo utra vaf var vj vjc vje vjs vmax vtf vto xcjc xj xqc xtb xtf xti
analysis	type parameters
option	name value
output variable	name value
error	name type status

Table 3.2: The basic data classes for the Expert Emulator

3.3 Rule Acquisition and Organization

Each rule in the Expert Emulator corresponds to a unit of human problem-solving expertise. Rules are added to the knowledge base as they are acquired in the following ways:

- from experienced SPICE users
- from circuit designers
- from simulation experts
- by experimenting with the simulator
- by monitoring the behavior and common errors of novice users of SPICE
- from the several classes of error checks that exist in SPICE; the error messages and warnings of SPICE2 are listed in Appendix B.

From an initial investigation of the various types of rules it has been concluded that the task of collecting a complete set of rules is a difficult undertaking. In addition, it has been felt that designers will always want to add their "own" rules that reflect their personal expertise and needs. There may be a need to modify certain rules, as a result of changes to the simulator (new versions). Hence, instead of attempting to compile a "complete" set of rules, the following objective was set:

Research Objective 2 *Investigation of techniques that would simplify further the incorporation of new rules to the knowledge base of* NECTAR *by individual users of the environment —a task already made quite straightforward by the choice of the production-system model.*

The outcome of the research toward the above objective is presented in Section 5.7.

Rules can be classified according to several characteristics, such as the type of simulation error, the type of circuit analysis, and the class of circuit. Experience from previous rule-based systems has shown that when the number of active rules is in the several hundreds, significant performance degradation occurs [Walters88]. Consequently, the following classification criterion has been chosen: *Rules are to be divided in rule sets so that for each application of the Expert Emulator only one rule set is active.* By deactivating rules unnecessary for a particular application, the number of active rules is reduced and the point of performance degradation becomes more distant.

Rules are first divided in three sets according to the point in the design cycle at which they become relevant (see Section 4.6).

- Rules for recovery from convergence and other simulation errors
- Rules for data checking before the simulation
- Rules for design aid (not related to simulation problems)

Each rule set can be subdivided in smaller sets when the need arises. In particular, the type of simulation error is a good classification criterion for the error-recovery set.

The remainder of this chapter contains examples of applying representative rules from each set on test circuits.

3.4 Overcoming Convergence and Other Simulation Problems

The Newton-Raphson algorithm is the backbone of the SPICE routines for nonlinear DC and transient analysis, where it is applied to systems of nonlinear algebraic equations[1]. An advantage of the Newton-Raphson algorithm is the existence of the following theorem that states the criteria under which the algorithm converges (and does so quadratically) [Forsythe77, Ralston78].

Theorem 1 (Convergence of Newton-Raphson algorithm) *Let the function $\mathcal{F}(x)$ be twice continuously differentiable and have a simple root for $x = x_r$. Then the sequence $x^{(i)}$, $i = 0, 1, 2, \ldots$, generated by the Newton-Raphson algorithm*

$$x^{(i+1)} = x^{(i)} - \frac{\mathcal{F}(x^{(i)})}{\mathcal{F}'(x^{(i)})}$$

will converge to the root x_r, provided that $x^{(0)}$ is sufficiently close to x_r.

The theorem can be generalized for the multidimensional case $\underline{\mathcal{F}}(\underline{x}) : \Re^m \mapsto \Re^n$, under the assumption that the Jacobian $\frac{\partial \underline{\mathcal{F}}(\underline{x})}{\partial \underline{x}}$ is Lipschitz continuous.

Most of the convergence problems in SPICE result from violations of the above convergence criteria. First, providing an initial guess close to the solution can be difficult.

[1] In transient analysis, a set of algebraic equations is derived from the integration of a set of differential equations.

Secondly, the model equations often have discontinuous derivatives [Vladimirescu80]. Some users get around the discontinuity problems by modifying the model equations, but one has to be careful about the consequences of such actions on the general behavior of the algorithms [Sangiovanni81]. Finally, the algorithm in SPICE is not the pure Newton-Raphson algorithm but a modified version. For instance, heuristics are used to limit the danger of numerical overflow because of the exponential characteristics of certain devices [Nagel75].

Some convergence problems with SPICE are related to other algorithms, such as the numerical integration algorithms, the time-step control algorithms, Muller's iteration method (applied in the calculation of poles and zeros), etc.

The Expert Emulator uses its knowledge base to infer reasons for nonconvergence. The essence of this phase is to recognize *patterns* that lead to convergence problems and employ special techniques to overcome the problems. The following is a list which is typical of the error-prone patterns recognized by NECTAR:

- forward-biased source/drain-bulk pn-junctions in MOS transistors
- rings of pn-junctions
- positive feedback loops
- regenerative switching circuits
- nodes isolated by high impedance
- erroneous or incomplete specification of connections between circuit elements
- unrealistic values for circuit parameters
- inappropriate values for SPICE control parameters
- insufficient use of simulator-control options (tolerances and limits)
- input-format violations.

Corrective techniques used by experts and employed in NECTAR include:

- correcting wrong connections
- adding parasitics
- modification of simulator-control parameters
  - using the OFF option for devices in the feedback path

  - changing ABSTOL (good values are: $.1\mu A$ for MOS and $1pA$ for bipolar)
  - increasing iteration limits

- modification of model parameters
  - specifying capacitance and resistance in models
- presetting the initial guess for Newton-Raphson (NODESET option)
- setting initial conditions (IC option)
- switch integration method, e.g., from trapezoidal to Gear's
- use of the source-stepping technique.

The following examples demonstrate the application of error-recovery rules. It should be noted that modifications to the input file are not only suggested by the rules but made automatically. However, the user has final control and can override any changes.

Example 1 (MOS Oscillator)

Figure 3.1 shows the schematic of a n-channel MOS relaxation oscillator. A novice user of SPICE might request a transient analysis, as indicated in the SPICE input file (Figure 3.2), but without specifying charge storage in the transistor model. A SPICE simulation on that file aborts before completion with the error message shown in Figure 3.3a. As a result of this, the rule set in the knowledge base of the Expert Emulator that handles time-step errors is activated. The following rule uses circuit-topology pattern matching, based on the net-list of the circuit[2].

```
(rule cross-coupled-mos
  (if
    (mos (name ?m1) (drain ?y) (gate ?x))
    (mos (name ?m2) (drain ?u) (gate ?v))
    (capacitor (terminals (?y ?v)))
    (capacitor (terminals (?x ?u))))
  (then
    (send-message "Positive feedback loop ?x–?y–?v–?u–?x detected.")
    (make (node (name ?x) (type possibly-isolated)))
    (make (node (name ?v) (type possibly-isolated)))))
```

[2] '?' characters in rules denote variables.

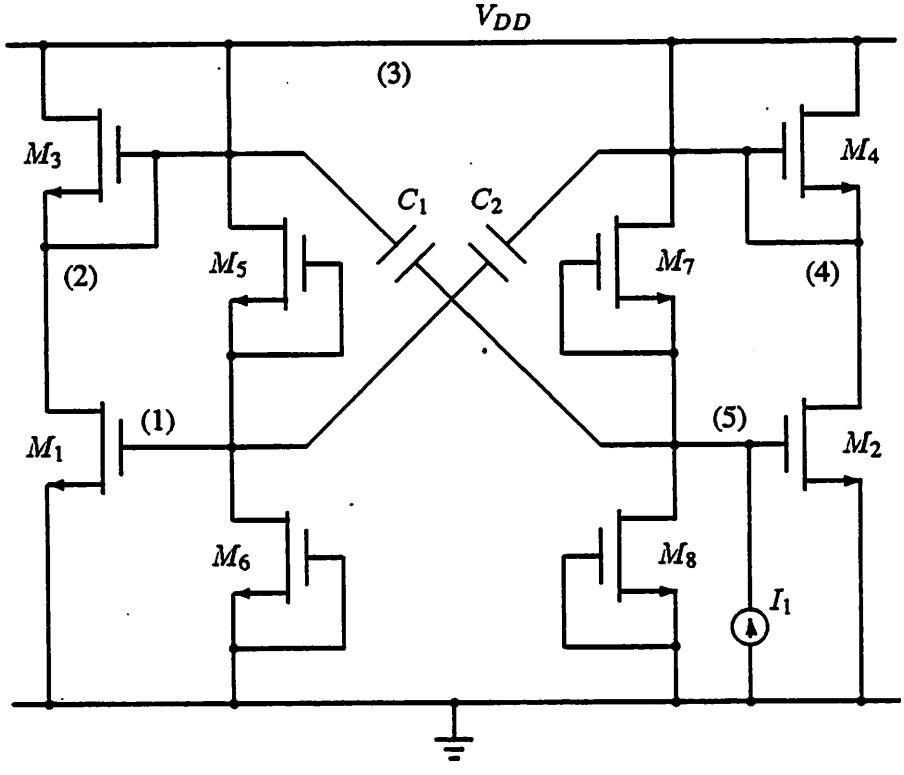


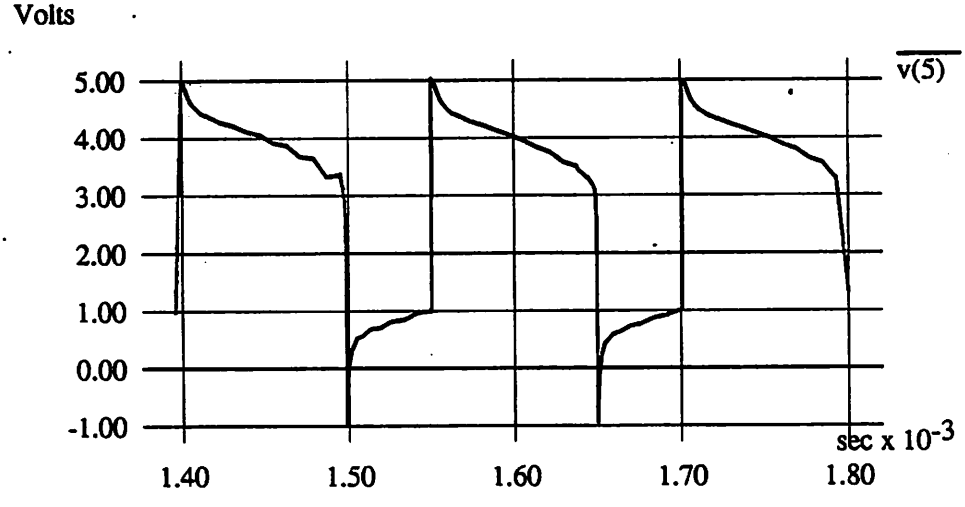
Figure 3.1: The MOS-Oscillator example

```
MOS relaxation oscillator
m1 2 1 0 6 mod1 w=100u l=5u
m2 4 5 0 6 mod1 w=100u l=5u
m3 3 2 2 6 mod2 w=20u l=5u
m4 3 4 4 6 mod2 w=20u l=5u
m31 3 1 1 6 mod2 w=20u l=5u
m10 1 0 0 6 mod2 w=20u l=5u
m35 3 5 5 6 mod2 w=20u l=5u
m50 5 0 0 6 mod2 w=20u l=5u
.model mod1 nmos vto=+0.7 kp=30u
.model mod2 nmos vto=-0.7 kp=30u lambda=0.01
c1 2 5 100p
c2 4 1 200p
vee 6 0 -9
vdd 3 0 5
i1 5 0 pulse 10u 0 0 0 0 1
.tran 2u 1800u 1400u
.plot tran v(5)
.width out=80
.option nopage nomod limpts=1001
.end
```

Figure 3.2: Input file for the MOS-Oscillator example

ERROR: INTERNAL TIMESTEP TOO SMALL IN TRANSIENT ANALYSIS

(a)



(b)

Figure 3.3: SPICE results for the MOS-Oscillator example: (a) initial run; (b) run after the addition of parasitic capacitors

This rule[3] recognizes the cross-coupled configuration of the four transistors M_1, M_2, M_3, and M_4, and the positive feedback loop around nodes 1–2–5–4–1. A second rule examines whether the two gate nodes in the loop (nodes 1 and 5) are dynamically isolated.

```
(rule isolated-node
  (if
    (node (name ?n) (type possibly-isolated))
    (not capacitor (terminals ?n 0)))
  (then
    (modify (node (name ?n) (type isolated))
    (make (capacitor (term1 ?n) (term2 0) (value 10fF)))
    (send-message "Parasitic grounded capacitor added at node ?n.")))
```

Node 1 is the gate of Transistor M_1. The model for M_1, mod1, does not include any charge storage. Hence, there is no capacitive path from Node 1 to ground through the transistor. In addition, there is no explicit path via external capacitors. Therefore, Node 1 is found by the rule to be dynamically isolated. The same is true for Node 5, the gate of M_2.

To achieve convergence, NECTAR adds small parasitic capacitors from the two gate nodes to ground. The value for these capacitors is set to 10 fF, which is small enough not to change significantly the operation of the circuit. The addition of the two capacitors is implemented as two extra lines in the input file:

```
c3445  1  0  1.0e-14
c3447  5  0  1.0e-14
```

NECTAR sends the modified input file to SPICE, which this time is able to converge. The results of the completed SPICE simulation are shown in Figure 3.3b.

Error Explanation

To see why SPICE aborted the first simulation run and why the parasitic capacitors helped in the second, consider the corresponding transient-simulation graphs of V_1, the voltage at Node 1, during a low-to-high transition. Assume that t_{n-1} is the last time-point for which SPICE computed zero voltage for V_1 (Figure 3.4). At first, SPICE computes the n^{th} time-point with a time-step equal to a computed initial value, TINIT. Because of the positive

[3] A similar rule applies to cross-coupled bipolar devices; in such cases, code replication could be avoided with the use of hierarchical data structures and property inheritance (object-oriented programming), which would allow a single higher-level rule ("cross-coupled-transistor").

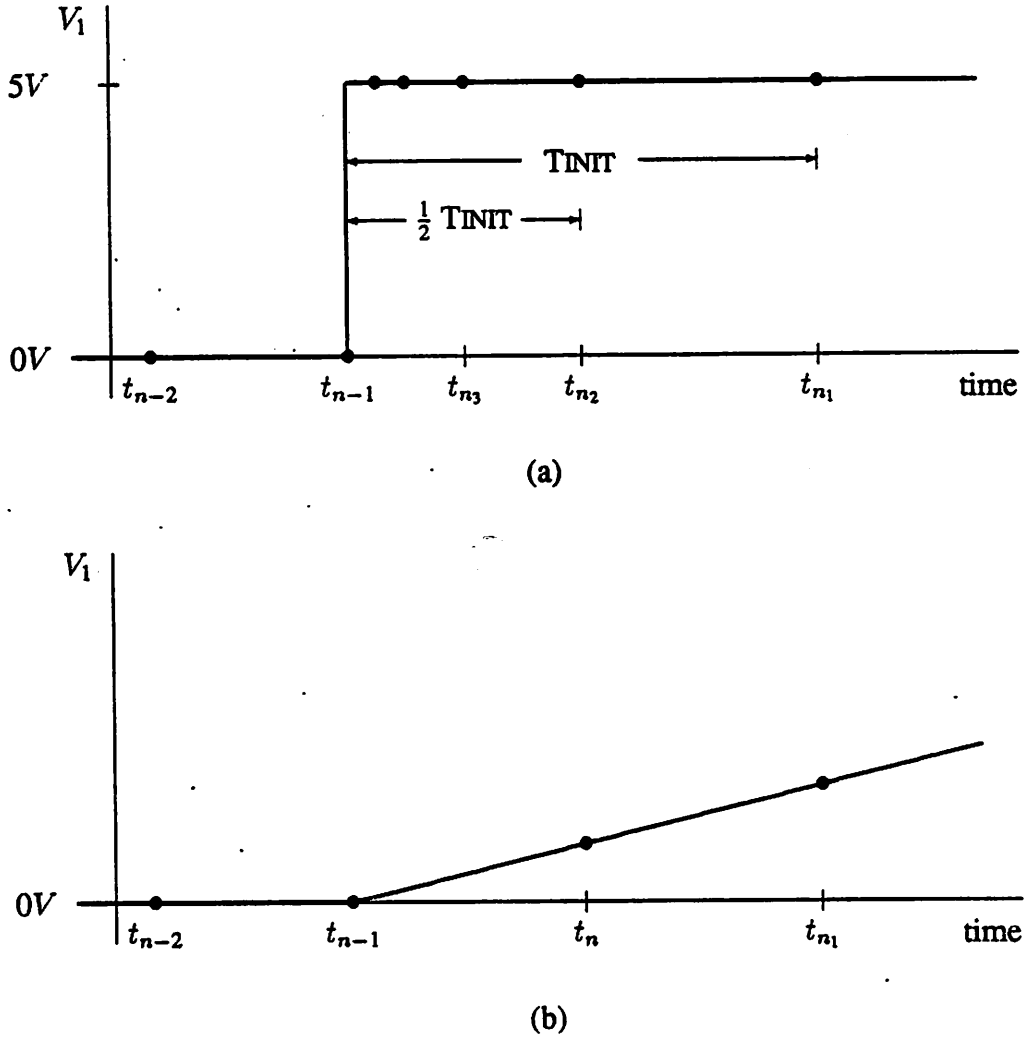


Figure 3.4: Explanation of the time-step error: (a) initial simulation run: because of dynamic isolation, the voltage at Node 1 jumps; (b) second run: the presence of a parasitic capacitor forces a smooth voltage rise

feedback, $V_1(t_n)$ tends towards infinity, but is limited to 5 Volts by the nonlinearities of the transistor. The time-step control algorithm in SPICE does not accept a solution with a 5-Volt jump from the previous time-point. The time-step is cut to half and a new calculation is performed for time-point t_{n_2}. Again, because of the positive feedback, $V_1(t_{n_2})$ is found to be 5 Volts. The time-step is cut in half again and the process continues until the time-step becomes less than TMIN, an internal SPICE constant. At that point the simulation run aborts.

The presence of C_p, the parasitic capacitor, in the second run prohibits voltage jumps at Node 1:

$$\left.\begin{array}{l} i_{C_p} < \infty \Rightarrow \exists k \in \Re : i_{C_p} < k \\ i_{C_p} = C_p \frac{dV_{C_p}}{dt} \end{array}\right\} \Rightarrow \frac{dV_{C_p}}{dt} < \frac{k}{C_p} \Rightarrow$$

$$\Rightarrow V_{C_p}(t) - V_{C_p}(t_{n-1}) < \frac{k}{C_p}(t - t_{n-1}) + \mathcal{O}((t - t_{n-1})^2)$$

It follows that, for realistic values of C_p, at some iteration of the time-step control algorithm the time-step, $t - t_{n-1}$, becomes small enough for the voltage increase, $V_{C_p}(t) - V_{C_p}(t_{n-1})$, to be less than the maximum accepted by the algorithm.

Example 2 (Pull-Up)

Figure 3.5 shows a digital circuit that models a pull-up load. A transient analysis using SPICE2 with the input shown in Figure 3.6 fails with a time-step error at $t = 1.88ns$. The partial results, shown in Figure 3.7a, suggest that the problem lies with the voltage feed-through from the gates of transistors M_1–M_5, which results in the forward biasing of the corresponding substrate junctions. The following rule[4] detects the suspect transistors and, to alleviate the problem, reduces the values of the gate capacitances by increasing the oxide thickness in the models [Meyer71], [Mayaram88, pages 203–204].

```
(rule mos-oxide-thickness
  (if
    (error (type timestep))
    (voltage-source (term1 ?t1) (term2 ?t2) (value pulse ?*))
    (mosfet (gate (or ?t1 ?t2) (model ?m))
    (model (name ?m) (tox ?tx:(< 1e − 7))))
```

[4] "?*" denotes a multiple-valued variable.

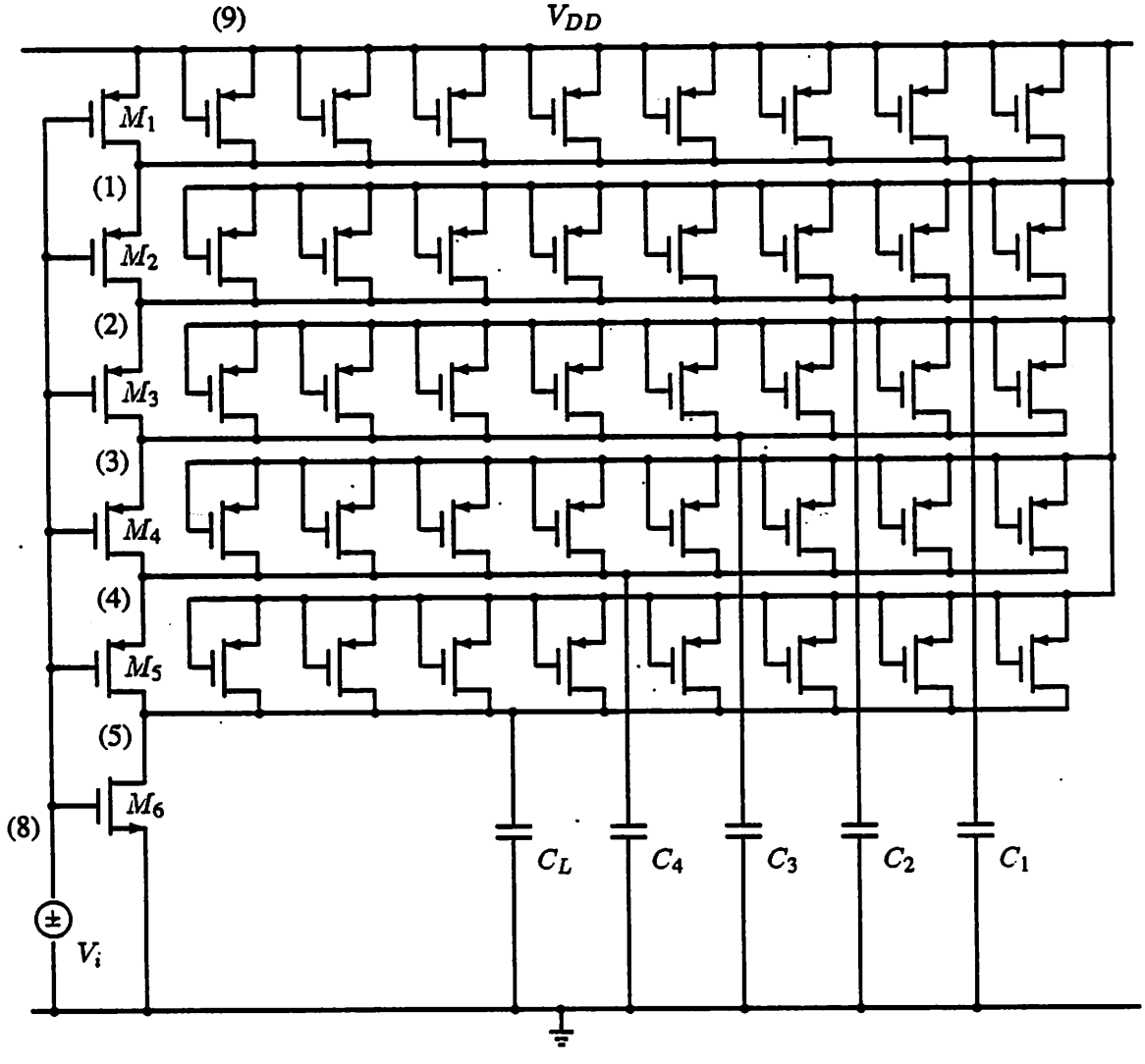


Figure 3.5: The Pull-Up example

```
Pull-up
.Model N NMOS level=2 vto=0.75 kp=76.0u gamma=0.4 lambda=0.025
+ tox=25n nsub=4e16 tpg=1 xj=0.45u ld=0.4u uexp=0.16 vmax=5.5e4
+ RSH=35 js=1u cgso=220p cgdo=220p cj=230u cjsw=260p cgbo=4p
.Model P PMOS level=2 vto=-0.75 kp=27.0u gamma=0.5 lambda=0.045
+ tox=25n nsub=2e16 tpg=-1 xj=0.4u ld=0.05u uexp=0.15 vmax=9.0e4
+ RSH=120 js=1u cgso=220p cgdo=220p cj=670u cjsw=215p cgbo=4p
VDD 9 0 DC 5
Vi 8 0 pulse(0 5 1ns 1ns 1ns 3ns 20ns)
m1 1 8 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m2 2 8 1 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m3 3 8 2 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m4 4 8 3 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m5 5 8 4 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m6 5 8 0 0 N l=1.6u w=14.4u ad=20.0p as=20.0p pd=12u ps=12u nrd=0.15 nrs=0.15
m11 1 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m12 1 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m13 1 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m14 1 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m15 1 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m16 1 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m17 1 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m18 1 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m21 2 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m22 2 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
:
m55 5 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m56 5 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m57 5 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
m58 5 9 9 9 P l=1.6u w=14.4u ad=34.6p as=34.6p pd=19u ps=19u nrd=0.17 nrs=0.17
CL 5 0 330ff
C1 1 0 1ff
C2 2 0 1ff
C3 3 0 1ff
C4 4 0 1ff
.tran 0.1ns 20ns
.print tran v(8) v(5)
.options limpts=2000
.options abstol=1n vntol=1u
.options defl=1.6u
.options itl1=2000
.width out=80
.END
```

Figure 3.6: Input file for the Pull-Up example

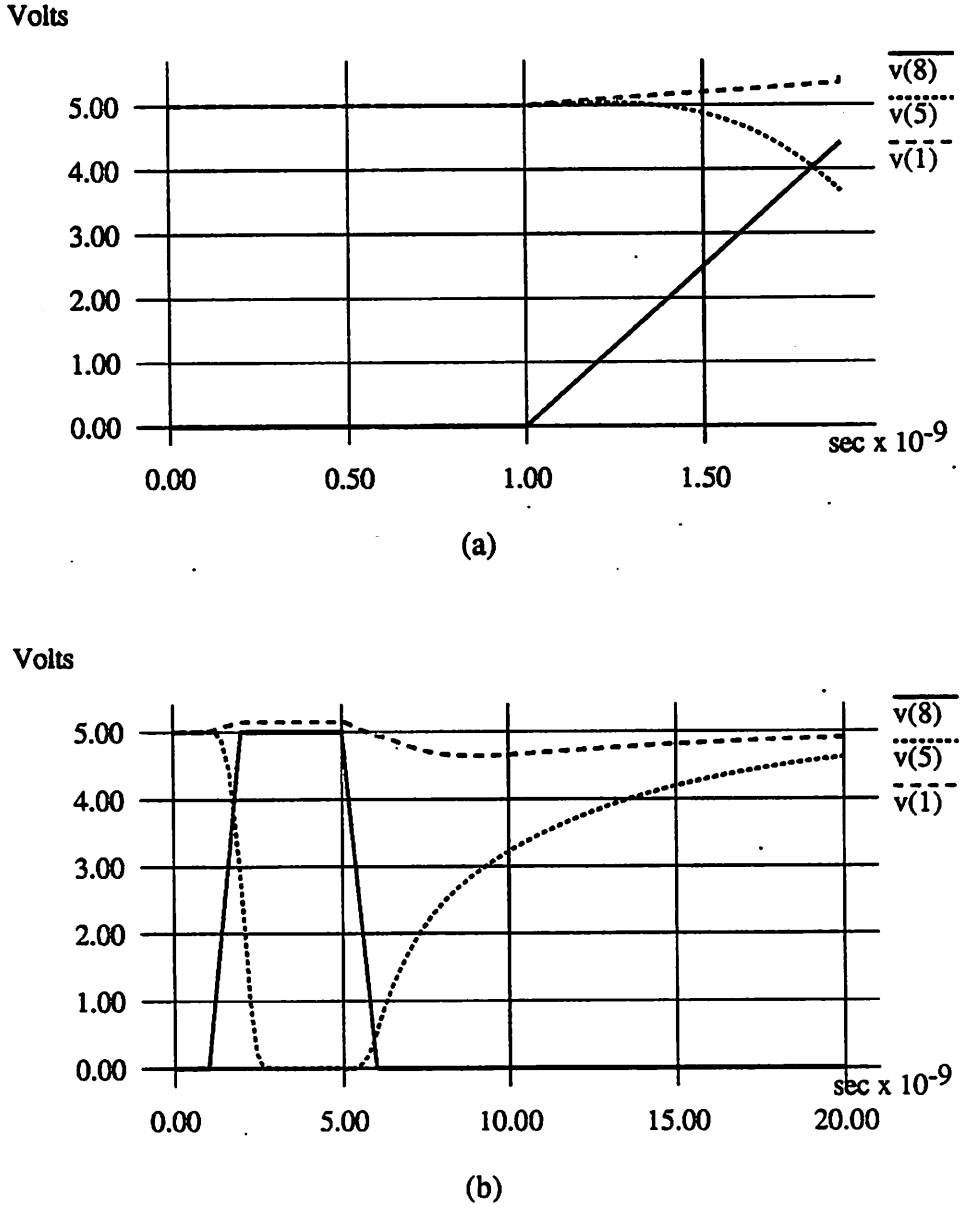


Figure 3.7: SPICE results for the Pull-Up: (a) initial run; (b) run after modification

```
(then
  (modify (model (tox 1e-7)))
  (send-message "Oxide thickness for model ?m set to 100nm.")))
```

A similar effect could be obtained with the artificial rise of the substrate voltages to a level that would prevent forward biasing. Figure 3.7b shows the results of the completed simulation after the modification of "tox."

Example 3 (Iteration Limit)

The following simple rules detect violations of the iteration limit in transient analysis. As suggested in the corresponding SPICE error message (Appendix B), the rules override this limit using the optional parameter ITL5.

```
(rule no-iteration-limit
  (if
    (error (type iteration-limit))
    (not (option (name itl5))))
  (then
    (make (option (name itl5) (value 20000)))
    (send-message "Iteration limit set to 20000.")))

(rule low-iteration-limit
  (if
    (error (type iteration-limit))
    (option (name itl5) (value ?v)))
  (then
    (modify (option (name itl5) (value (* 8 ?v))))
    (send-message "Iteration limit set to " (* 8 ?v) ".")))
```

In the MOS-Oscillator above, the iteration limit is violated when the parasitic capacitors are introduced. The completed simulation in Figure 3.3b is achieved after the application of the rule "no-iteration-limit."

Example 4 (Bipolar Oscillator)

The bipolar (blocking) relaxation oscillator shown in Figure 3.8 — its SPICE input is in Figure 3.9 — is another example of an error in transient analysis. Figure 3.10a shows the partial simulation results up to the time-step error at $t = 1.87\mu s$. The existence of a resistorless feedback loop (2–L_1–L_2–3–2) is one possible explanation for this error,

which is not well understood. Nevertheless, the introduction of a resistive element in the loop (in this case, using the base-resistance parameter of the bipolar model) corrects the problem, as shown in the results of Figure 3.10b[5]. The error is detected and amended with the application of the following rule:

```
(rule bjt-base-resistance
  (if
    (error (type timestep))
    (bjt (model ?m))
    (model (name ?m) (rb 0)))
  (then
    (modify (model (rb 100)))
    (send-message "Base resistance for model ?m set to 100 ohms.")))
```

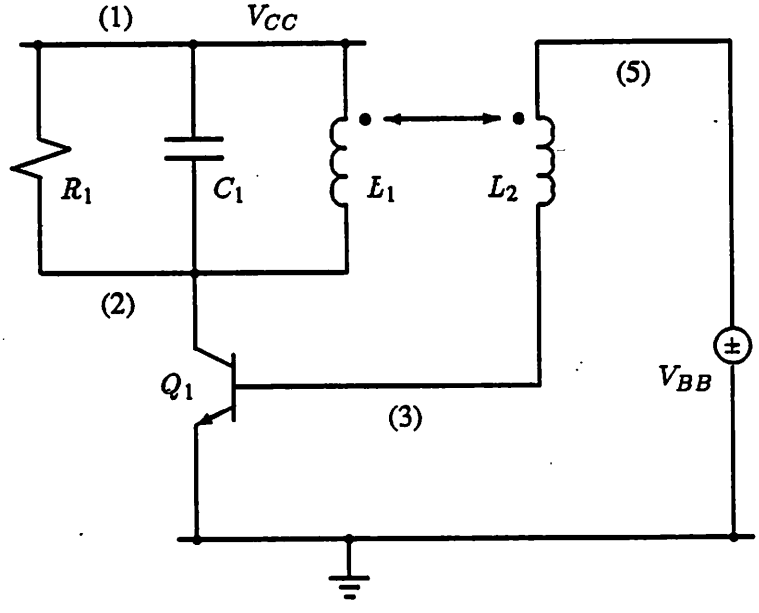


Figure 3.8: The Bipolar-Oscillator example

[5]The simulated collector voltage ($> 60V$) may not be practically feasible.

```
Bipolar oscillator
vcc 1 0  10
r1 1 2 1k
c1 1 2 300p
l1 1 2 3u
l2 3 5 20.83n
k1 l1 l2 1
vbb 5 0 pulse( .708 .76 5e-9 0 0 1e-3)
q1 2 3 0 mod1
.model mod1 npn bf=100 is=1e-16
.tran 20e-9 4e-6
.plot tran v(2) (-10,20)
.options nopage
.width out=80
.end
```

Figure 3.9: Input file for the Bipolar-Oscillator example

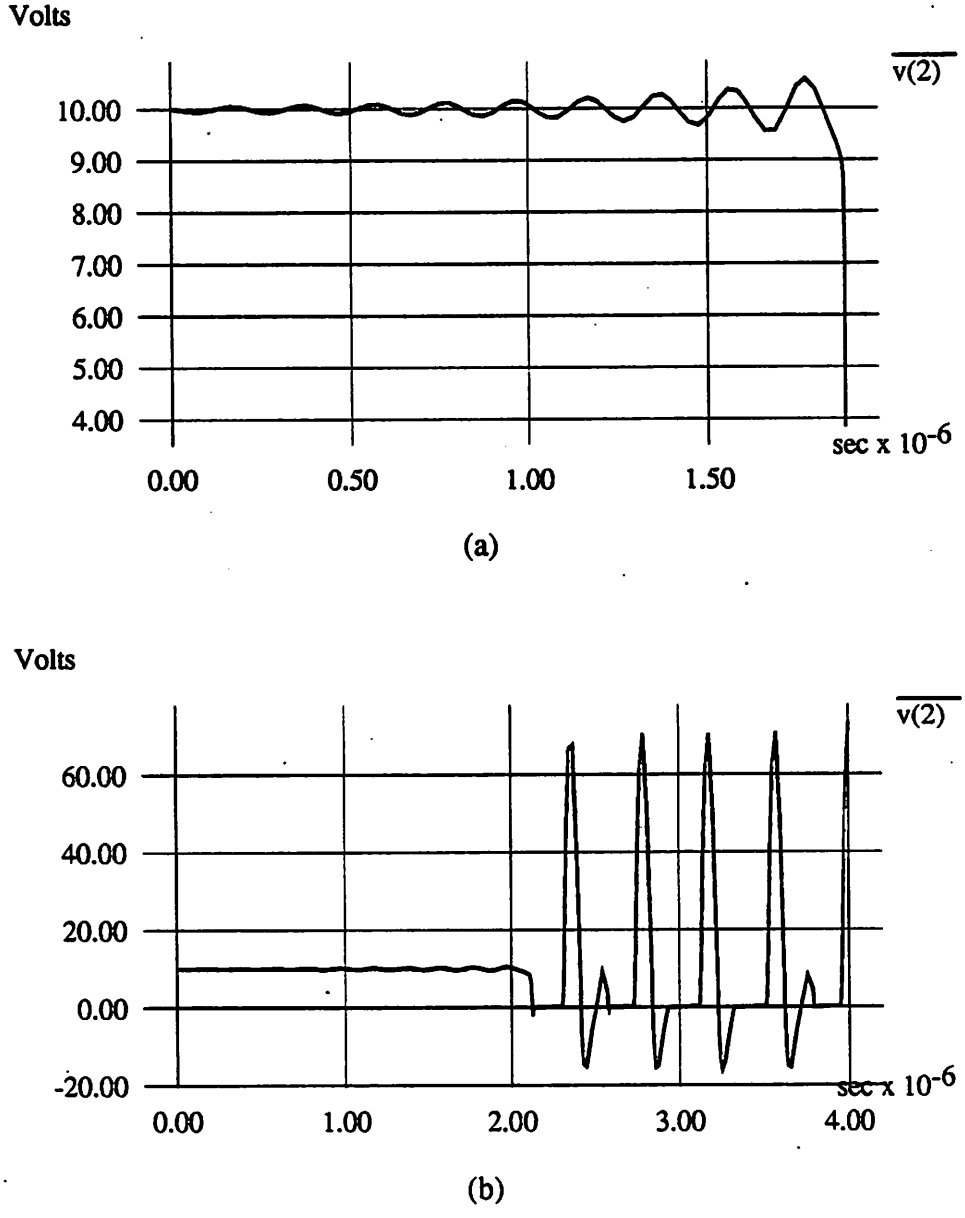


Figure 3.10: SPICE results for the Bipolar Oscillator: (a) initial run; (b) run after modification

3.5 Data Checking Before the Simulation

As mentioned above, the input to the Expert Emulator comes from the input and output SPICE data. It is possible, however, to identify problems by looking at the SPICE input data alone. Such checks can be made before the simulation, since no simulation results are used, and are similar to checks that exist hard coded in SPICE2 and SPICE3. The rules for presimulation checks are similar to convergence-error rules. The only difference lies in the absence of simulation-result patterns in the LHS of presimulation rules.

The application of this type of rule is illustrated with the following example.

Example 5 (Enhancement-Load Inverter)

Consider the enhancement-load inverter circuit, whose schematic is shown in Figure 3.11. Without Transistor M_1 and Capacitor C_1, the maximum value for the output

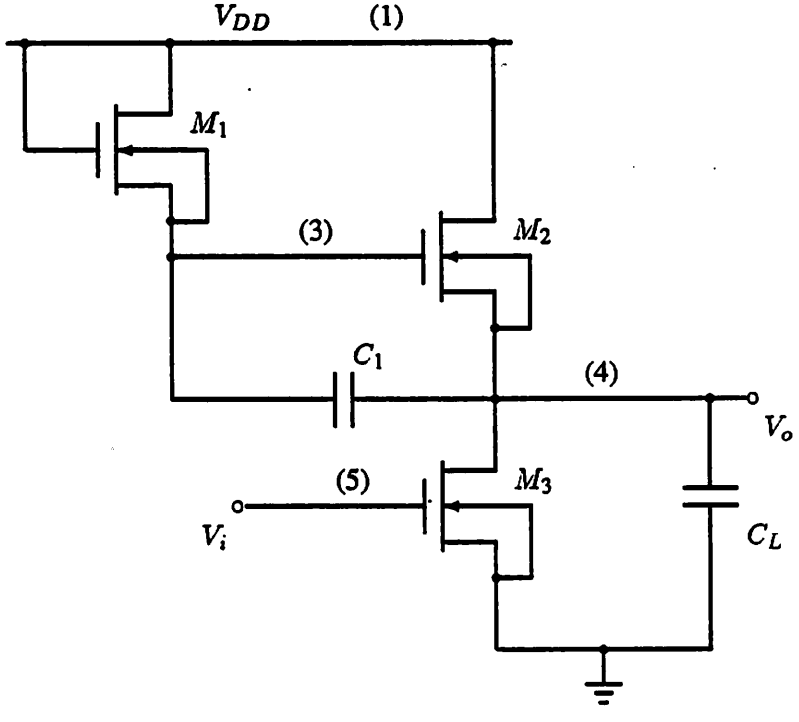


Figure 3.11: The Enhancement-Load-Inverter example

voltage V_o would be $V_{DD} - V_t$. By adding M_1 and C_1, the designer ingeniously raises the gate voltage of M_2, so that V_o can reach V_{DD}. This dynamic design technique, known as

bootstrapping, eliminates the need for an additional supply [Hodges88]. However, in the SPICE description shown in Figure 3.12, the substrate of M_1 is tied to the source. As a

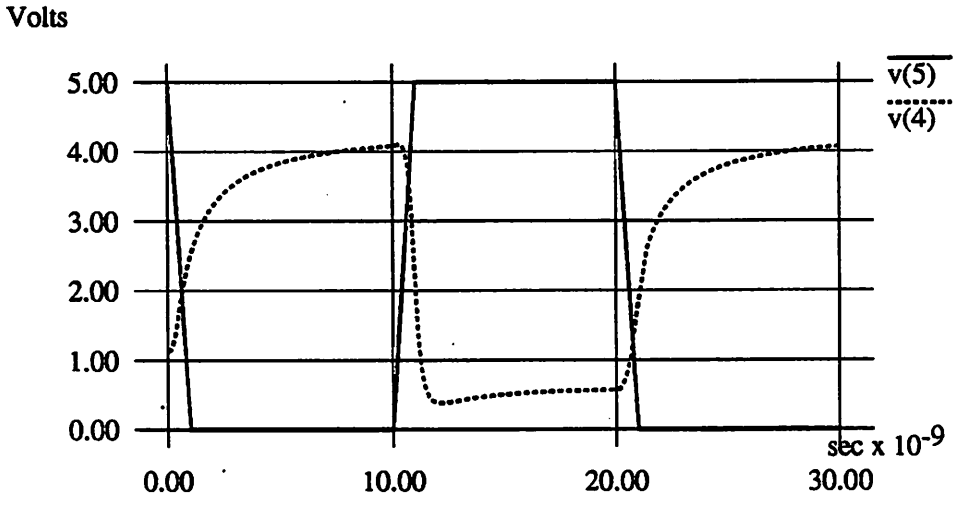
```
Enhancement load inverter
vdd 1 0 5.0
vin 5 0 pwl 0 5 1n 0 10n 0 11n 5 20n 5 21n 0
m1 1 1 3 3 mod2 w=5u l=3u
m2 1 3 4 4 mod2 w=5u l=3u
m3 4 5 0 0 mod2 w=5u l=3u
cl 4 0 .1pf
c1 3 4 .1pf
.model mod2 nmos level=5 vto=1.13 tox=0.050u nsub=2.5e16 uo=800
+ld=0.4u gamma=1.34 phi=0.75 nfs=5e10
+vmax=85k neff=2
+cgso=276p cgdo=276p
+cj=320u mj=0.5 cjsw=900p mjsw=0.33
+js=100u tpg=+1 xj=200n
.tran .2ns 30ns
.options reltol=1e-5
.print tran v(5) v(4)
.width=80
.end
```

Figure 3.12: Input file for the Enhancement-Load-Inverter example

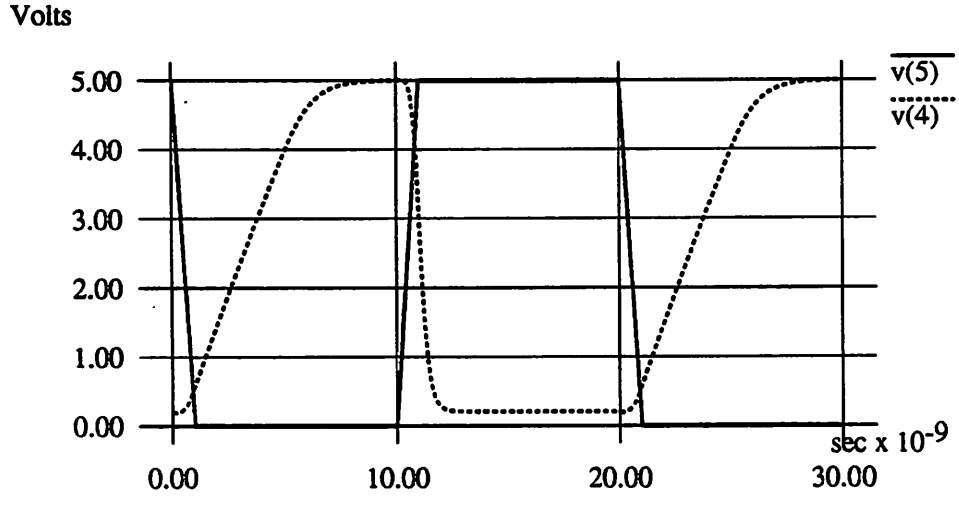
result, the substrate-drain junction of M_1 becomes forward biased limiting the voltage rise of Node 3 and, hence, the output swing, as shown in Figure 3.13a.

The following NECTAR rule checks whether, for digital circuits, the substrates of n-channel devices are connected to the most negative node (a dual rule checks p-channel devices).

```
(rule nmos-bulk-connection
  (if
    (mosfet (name ?t) (model ?m) (bulk ?b))
    (model (name ?m) (type nmos))
    (node (name ?n) (type most-negative))
    (not (equal ?b ?n)))
  (then
    (modify (mosfet (bulk ?n)))
    (send-message "Substrate of ?t connected to node ?n (most negative).")))
```



(a)



(b)

Figure 3.13: SPICE results for the Enhancement-Load Inverter: (a) with wrong substrate connections; (b) with corrected connections

The rule connects the substrate of M_1 to ground and the desired output swing is obtained, as illustrated in Figure 3.13.

3.6 Rules for Design Aid

This class of rules for NECTAR involves circuit design constraints. These rules are activated, regardless of convergence problems, when requested by the user (see Figure 4.7 in Chapter 4), to satisfy design specifications. A typical example might be to provide a minimum gain for a MOS amplifier stage with respect to device dimensions.

A novice circuit designer often attempts to meet the specifications using a "trial-and-error" procedure. In the process, SPICE is run numerous times, each time with a different value for some parameter. This procedure is often done randomly and uses significant amounts of CPU time.

NECTAR can accept and apply circuit design rules and formulas to modify appropriate circuit parameters and achieve the desired functionality. Such rules apply to specific circuit topologies. The design constraints are obtained from the analysis of similar circuits. It should be pointed out that these rules do not involve optimization.

The following example illustrates the use of design-aid rules.

Example 6 (Near-Sinusoidal Oscillator)

A bipolar near-sinusoidal transformer-coupled oscillator circuit is shown in Figure 3.14 — the input file is shown in Figure 3.15. This circuit may exhibit the phenomenon of "squegging" [Mayaram87], a multimode oscillation illustrated in Figure 3.16a. Analysis of the oscillator results in the criterion

$$C_e < \frac{2nC}{\left(\alpha - \frac{1}{n}\right)}$$

[Pederson90b] for the coupling capacitor C_e to avoid "squegging." This criterion has been coded in the following rule.

```
(rule near-sinusoidal-oscillator
  (if
    (bjt (collector ?c) (emitter ?e))
```

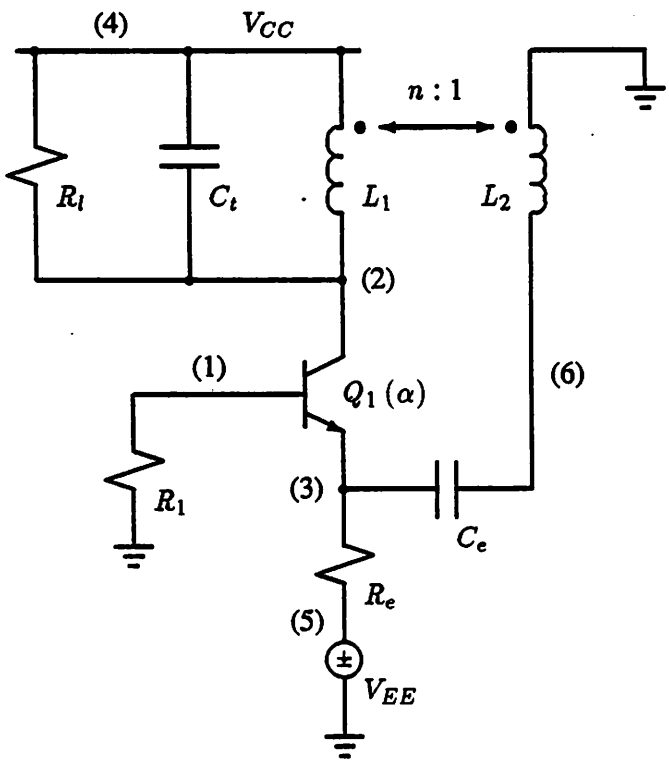


Figure 3.14: The Near-Sinusoidal-Oscillator example

```
Transformer-coupled oscillator
r1 1 0 1
Q1 2 1 3 mod1
vcc 4 0 10
rl 4 2 750
ct 4 2 450e-12
l1 4 2 5e-6
l2 0 6 0.05e-6
k1 l1 l2 1
ce 6 3 10e-9
re 3 5 4.65e3
vee 5 0 -10 pulse -15 -10 0 0 0 6e-6
.tran 30e-9 6e-6
.plot tran v(2)
.model mod1 npn is=1e-16 bf=100 rc=10
.options nopage nomod limpts=500
.width out=80
.end
```

Figure 3.15: Input file for the Near-Sinusoidal-Oscillator example

```
  (capacitor (name ?ce) (terminals ?z ?e))
  (capacitor (name ?ct) (terminals ?a ?c) (value ?v))
  (inductor (name ?l) (terminals ?a ?c) (value ?l1))
  (inductor (name ?m) (terminals ?z ?l2))
  (mutual-inductance (inductor1 ?l) (inductor2 ?m) (turns-ratio ?r)))
(then
  (modify (mutual-inductance (turns-ratio (sqrt (/ l1 l2)))))
  (modify (capacitor (name ?ce) (value (/ (* ?r ?v) (- 1 (/ 1 ?r))))))
  (send-message "Coupling capacitor ?ce changed to ?v.")))
```

The rule automatically modifies the value of C_e. Figure 3.16b shows the SPICE results for the new capacitor value.

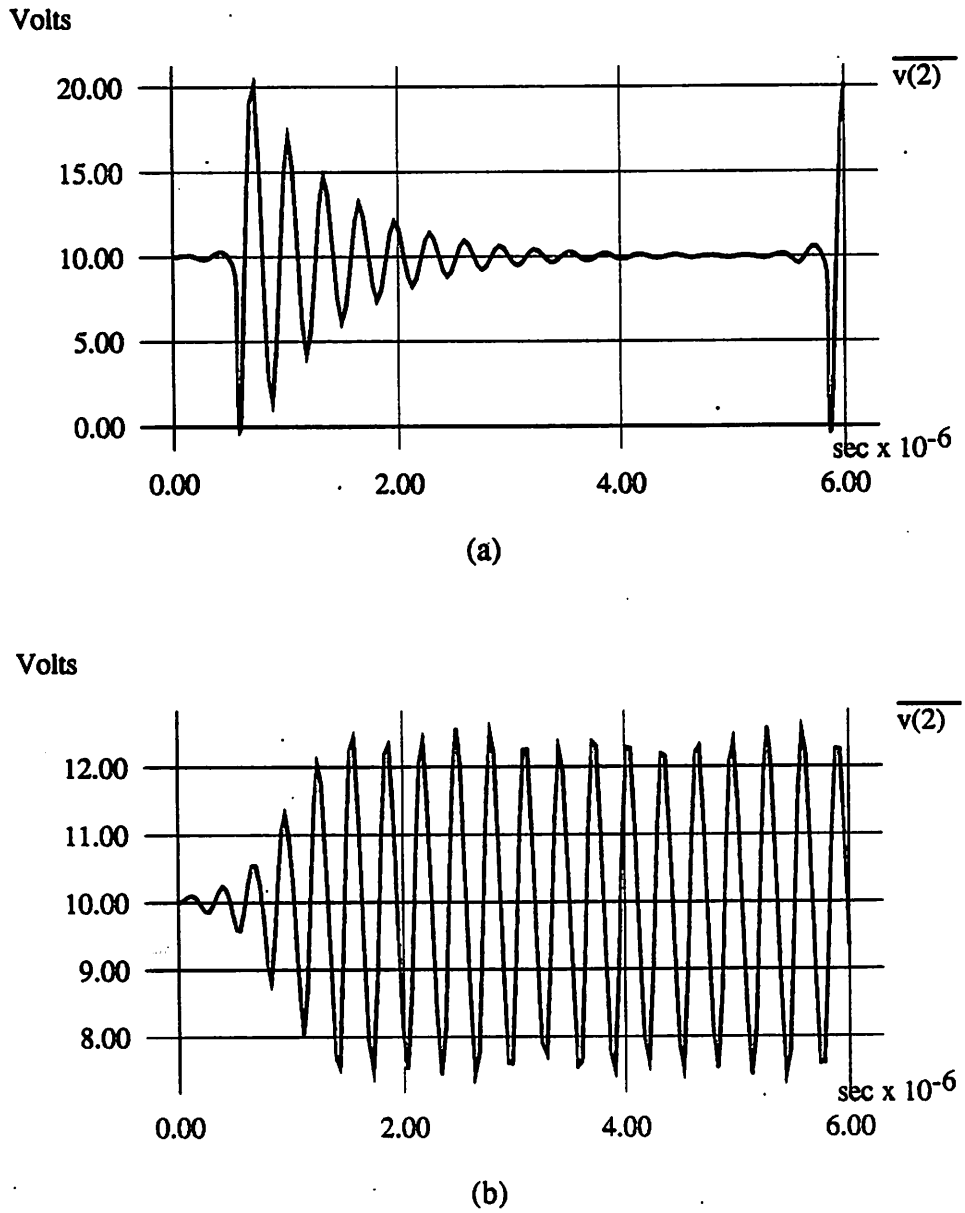


Figure 3.16: SPICE results for the Near-Sinusoidal Oscillator: (a) initial run; (b) run after modification

Chapter 4

Integration of Analog Verification Tasks in a Framework

4.1 Overview

In this chapter, NECTAR is expanded into a framework that integrates several analog verification tools running in a distributed computing environment. The user interface of the framework is presented in Chapter 5, and the actual implementation is described in Chapter 6.

NECTAR is generalized from the two-program environment presented in Chapter 2 to an open CAD framework for verification. The tools that are integrated in the framework include various simulation programs, simulation-result post-processors, design data editors, programs to check input data and recover from simulation errors, and auxiliary shell utilities. The close coupling of the tools allows the end user of the framework to focus on the design tasks, instead of on how to use the tools.

Design and simulation data is managed automatically by the framework. Data-storage formats have been chosen to minimize format conversions. In the presence of a distributed computing environment, NECTAR can direct jobs to remote machines, hidding communication details. Although the overall framework control lies with the user, a principal control cycle is prescribed to correspond to the main activity loop during analog design.

4.2 Evolution of a Framework

In Section 2.4 of Chapter 2, NECTAR is presented as the integration of two interacting programs, SPICE and the Expert Emulator. This section describes the generalization of NECTAR into a CAD environment that includes several simulation-related programs.

4.2.1 A First Generalization Step

As brought out in Section 2.4 concerning Option O–3 of Fact 1, a knowledgeable user might want to choose between several versions of SPICE, because different versions are best for the simulation of different types of circuits or for different analysis requests. As an example, early releases of SPICE3 lack a reliable distortion analysis capability. Such a capability is available in SPICE2G.6. Hence, users of SPICE3 turn to SPICE2 when the need for a distortion analysis arises. The decision to choose a different (than the default) version of the simulation program may come either after an initial unsuccessful run with the default version or following an examination of the analysis requests before any simulation run.

The process of choosing among several simulators can be automated using NECTAR. The augmented environment, shown in Figure 4.1, comes from the scheme in Fig-

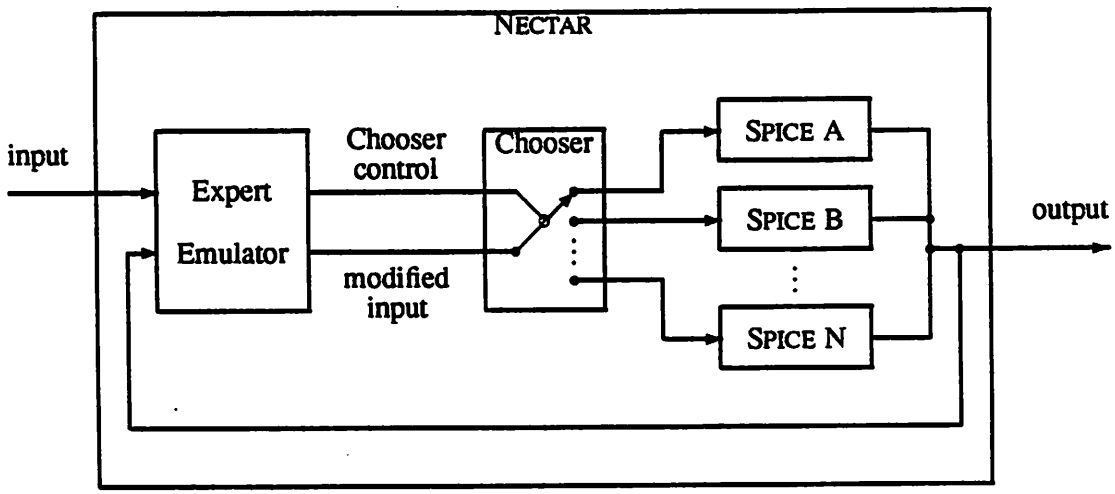


Figure 4.1: Choosing from several SPICE versions

ure 2.3 with the following modifications: instead of just one, several SPICE versions are

embedded in NECTAR; expert knowledge about the analysis capabilities and limitations of various versions of SPICE is incorporated in the Expert Emulator as rules; a choosing mechanism, controlled by the Expert Emulator, runs the appropriate SPICE version on the input data.

At this point, NECTAR integrates more than two programs. Further additions and enhancements have extented NECTAR into a CAD *framework* for analog circuit verification, as described below.

4.2.2 CAD Frameworks

Circuit designers are becoming increasingly dependent on a multitude of CAD tools from various sources to speed up the design process, to manage the growing amounts of design data, and to accommodate the strengthening interdependencies between technologies, design styles, and design teams. The concept of a CAD framework has emerged recently to meet the need for integration and automatic management of today's numerous and polymorphic CAD tools.

A CAD framework can be defined as a software infrastructure that provides a common operating environment for CAD tools. The infrastructure can include user interface, tool interaction, interprocessor communication, and data management facilities. These and other functions of CAD frameworks are necessary to tie together not only different tools but also different processors, different operating systems, and different human designers. A prime goal of CAD frameworks is to allow the end users (designers) to focus on the design activity by shielding them from tool and system details.

Some of the main issues in the development of CAD frameworks are the types of tools to be integrated, the representation scheme for design data, the representation of design knowledge, and the control of the design process. These issues have been addressed in several research efforts in the field. Designer's Workbench [Friedenson82], one of the first CAD frameworks, integrated already existing tools under a central user interface. The Palladio project [Brown83] introduced a hierarchical design representation integrating tools built specially for that representation. In DEMETER [Siewiorek84], a commercial database was used for storage of information common to four different tools running on different comput-

ers. The ongoing ADAM project [Granacki85] uses separately stored design knowledge to manipulate hierarchically represented design data. The Oct approach [Harrison86] provides a general hierarchical data management model shared by all integrated tools, which must be consistent to a formal set of data types and access parameters, called *policy*. ULYSSES [Bushnell87] integrates several dissimilar tools into a single conceptual framework based on a global data space for storage of intermediate results and goals, called *blackboard*, and special high-level representations of tool execution sequences, called *scripts*. Cadweld [Daniell89], a continuation of the ULYSSES project, decentralized the part of the control mechanism that is specific to each tool using object-oriented programming techniques [Stefik86].

Most of the frameworks above are concerned with all phases of IC design and, as such, must accommodate a host of different tools in several different areas:

- synthesis: high-level, logic, physical (placement and routing), etc.
- optimization: minimization of logic and finite-state machines, layout compaction, device sizing, etc.
- verification: circuit, device, timing, and switch-level simulation, design critiquing, etc.
- editing: schematic, layout, etc.
- translation: extraction, etc.

In each of the areas above, the problem solved varies considerably from the rest and applies to different aspects of the design data. The corresponding tools also differ considerably, thus complicating their interaction in a framework.

4.2.3 Analog CAD

In contrast to tools for the design of digital circuits, analog CAD tools are few and comparatively primitive. This disparity can be attributed to the fact that, compared to analog, digital circuits are characterized by higher topological regularity and are easier to partition into subcircuits, to analyze, and to design [Carley88]. This has led to a better understanding of digital circuits and the development of many digital CAD tools that

have contributed significantly to the more recent designs. In addition, digital circuits have achieved much higher chip densities, partly a reflection of the tremendous impact of the Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET), which is suited to realize digital functions more effectively than analog functions [Hodges88]. Digital circuits have attracted most of the attention of CAD-tool researchers and developers in the last several years.

Despite the lesser emphasis given on them, analog circuits are important for high-frequency applications, integrated sensors, real-time control systems, in realizing power supplies, and for interfaces between digital systems and external (naturally continuous) signals [Brodersen84]. Such interfaces usually are implemented with separate analog components and IC's. Recently, however, designers have started integrating both analog and digital subcircuits on the same chip. Even though the analog parts typically occupy about 10 percent of the chip, they now require about 90 percent of the total design time [Pederson90]. As a result, the need for more sophisticated analog CAD tools has become evident [Allen86]. The programs IDAC [Degrauwe87], OASYS [Harjani87], and OPASYN [Koh90] are some of the first analog synthesis tools.

Since analog synthesis tools are still in an experimental stage, analog design is, for the most part, an iterative "trial-and-error" process, as illustrated in Figure 4.2. First, the design specifications, i.e., the functional, performance, and physical properties of the circuit, are set. Then, potential circuit realizations are successively modified and checked, resulting in a gradual refinement of the design, until the specifications are met. The modification (editing) part of the design process is based on certain techniques, the most common of which is the **manual** input of design expertise by the designer. Once the topology of the circuit is fixed, the much smaller subspace of all circuits differing only in the values of certain parameters can be searched with optimization tools [Nye88, Shyu88]. As for the verification part of the process, analog circuit designers, including the designers of critical digital building blocks, primarily depend on electrical circuit simulation.

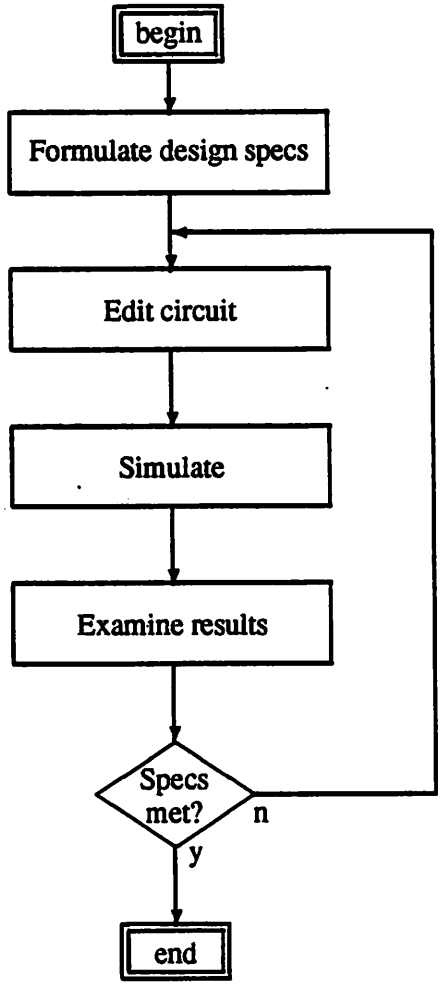


Figure 4.2: The iterative nature of analog-circuit design

4.2.4 The NECTAR Framework

The polymorphism of CAD tools constitutes not only a motive for integration but also an obstacle in the development of frameworks. In addition, the requirements for CAD frameworks are changing as tools and designs are changing and, hence, designing a framework is a task with a moving target. Therefore, attempting to develop a single framework for all CAD activities may not be the best approach. In this research, the scope of a CAD framework has been narrowed:

Research Objective 3 *Development of a framework for simulation and other tasks pertinent to the analog design cycle (Figure 4.2).*

The NECTAR framework evolved from the two-program shell of Figure 2.3 and the several-program environment of Figure 4.1 to the schema of Figure 4.3, which shows the various programs, databases, and hardware components that NECTAR ties together. Integrated programs include several simulators, post-processors of simulation results, design-data editors, the Expert Emulator, a special rule editor (presented in Section 5.7), and system utilities. There are two databases, one for design data and one for rules containing simulation and design expertise. NECTAR interfaces to different types of machines and displays. Finally, a user-interface module gives the NECTAR user direct control over the actions to be taken, including overwriting decisions by the Expert Emulator.

Since the focus is on analog verification, the various components of NECTAR do not need to be as abstract as those of a general CAD framework; thus, they can be made efficient. The goals of the NECTAR framework, including some goals common to general CAD frameworks, are the following:

- Collecting and making available in a single software shell an array of programs and functions related to circuit verification
- Development of a uniform user interface for all integrated tools
- Elimination of redundant information flow to and from the user
- Use of a common database for all tools
- Minimization of data-format translations
- Hiding tool and system details from the user

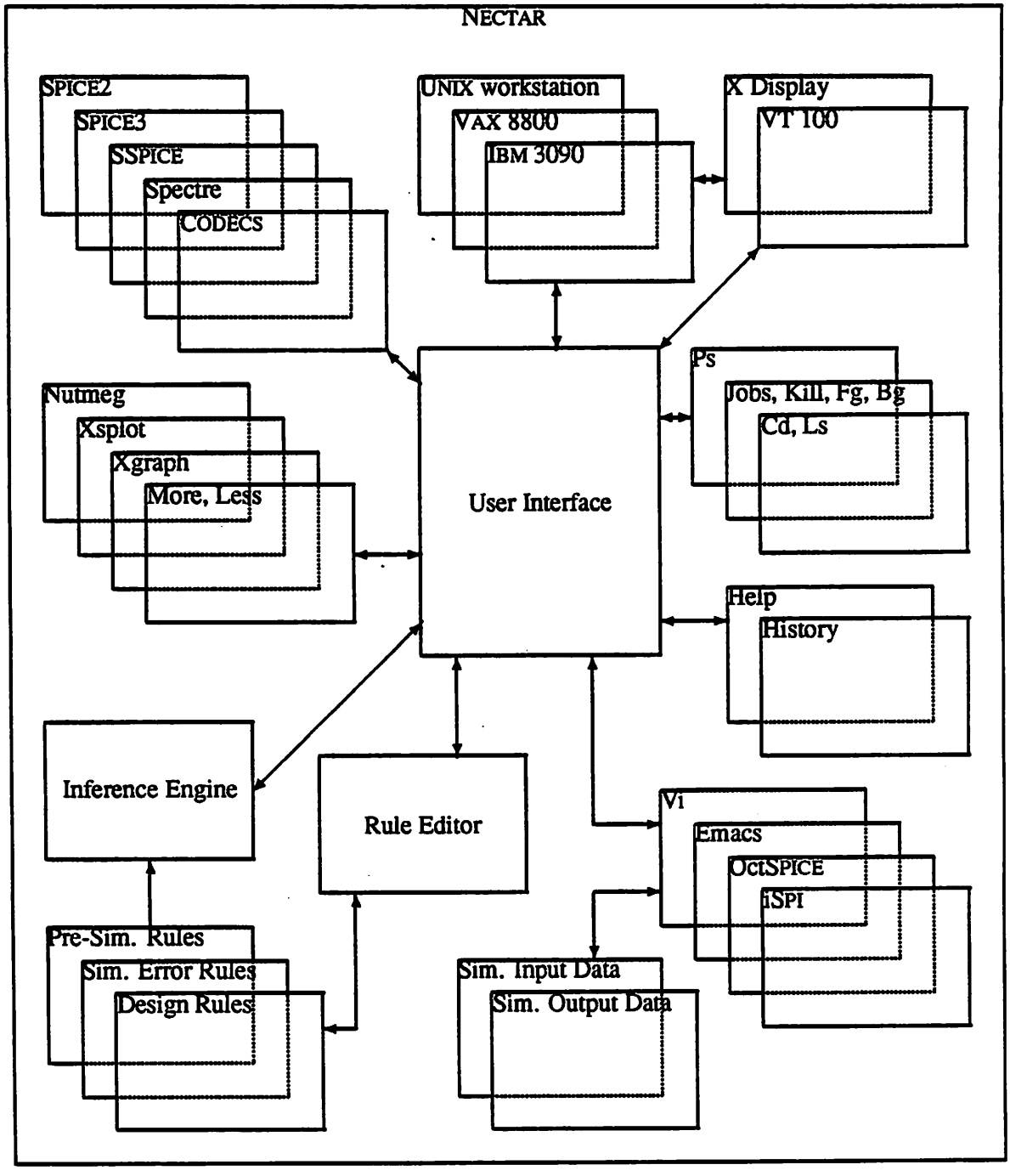


Figure 4.3: General view of the framework: programs, databases, and hardware are integrated in an environment for simulation

- Putting emphasis on design tasks, not tools
- Automatic execution of tasks needed most of the time
- Increasing the designer's productivity
- Minimization of errors by the user
- Minimization of training requirements for novice designers.

The remainder of this chapter describes the choices made regarding supported tasks, integrated tools, data organization, computing facilities, and framework control. Decisions for goals related to the user interface are presented in Chapter 5.

4.3 Tasks Supported by the Framework

NECTAR supports design tasks that normally are needed during the analog design cycle. For each task, one or more tools have been integrated in the framework. Multiple tools for a single task are included when no one tool covers all the requirements or when users' preferences are divided among several tools.

The following tasks can be accomplished using NECTAR:

Direct-Method Electrical Circuit Simulation: Various types of circuit simulators analyze circuits at different detail levels and, therefore, are characterized by varying run-time and result-accuracy properties. For most analog circuits, the optimal trade-off point between speed and accuracy corresponds to direct-method electrical-level simulation. Devices are represented at this level with analytical models relating their terminal voltages and currents. The circuit equations are solved directly to yield the voltage waveforms at all nodes and the current waveforms through the branches of the circuit. NECTAR gives access to several versions of the program SPICE varying in their analysis capabilities, algorithms used, or in the language in which they were programmed (FORTRAN for SPICE2; C for SPICE3), the compiler used (public domain or commercial), or the type of machine on which they run (UNIX workstations, VAX minicomputers, IBM mainframes).

Mixed-Level Simulation: The circuit properties in certain designs depend heavily on the

performance of one or more critical devices. There is a need to simulate critical devices with an accuracy higher than that of an electrical-level simulator. Device-level simulators [Price82] analyze devices by solving the nonlinear partial differential equations in space and time (Poisson's and current-continuity equations) obtained from device physics. It is assumed that the terminal voltages, which constitute the boundary conditions for the differential equations, are known or set. However, it is not easy to predict the boundary conditions for the realistic case of a device embedded in a circuit.

Mixed-level simulators, such as MEDUSA [Engl82] and CODECS [Mayaram88], combine circuit and device simulation. Critical devices are simulated at the device level with boundary conditions determined from the rest of the circuit, which is analyzed with an electrical circuit simulator. Mixed-level circuit and device simulators are much slower than electrical simulators. The program CODECS has been integrated in NECTAR. Even though it runs on smaller machines, CODECS jobs are usually submitted to a mainframe, because of the high CPU-time requirements.

Steady-State Simulation: The periodic steady-state behavior is of primary importance for large classes of circuits, such as amplifiers, oscillators, filters, multipliers, mixers, etc. Harmonic distortion, noise, power dissipation, gain, and other useful circuit parameters are calculated based on the steady state. However, steady-state simulations using direct time-domain methods can be computationally expensive, in particular for high-Q and narrow-band circuits.

Special simulators for the steady state use algorithms that bypass initial transients and compute the periodoc steady state directly. This results in faster simulations. The steady-state simulators Spectre [Kundert86] and SSPICE [Ashar89] have been integrated in NECTAR. Spectre simulates nonautonomous weakly nonlinear circuits in the frequency domain based on the method of *harmonic balance*. SSPICE, on the other hand, is able to simulate both autonomous and nonautonomous circuits in the time domain based on *shooting methods*.

Simulation-Result Post-Processing: After a simulation run is completed, usually a long stream of results is produced. Post-processors are programs that take the simula-

tion results and present them in such a form that the designer can easily evaluate the simulated circuit performance and compare it against the design specifications. Typically, a post-processor allows a user to plot voltage and current waveforms as well as measure certain quantities directly on the plots. By taking advantage of modern multiple-color bit-mapped displays, post-processors allow users to focus quickly on the data of interest.

Simulation programs often include their own built-in post-processors. Other simulators generate results in formats suitable to be processed by stand-alone post-processors. The choice of a post-processor often is a matter of personal taste. Hence, the goal for NECTAR has been to allow users to process and view results from any simulator using any one of several post-processing and plotting programs integrated in the framework. The available choices include the programs Nutmeg [Christopher87], Xsplot [Bradley87], Xgraph [Harrison88], and the text pagers More and Less. The data-format conversions between simulators and post-processors are made automatically, as described in Section 4.4.

Design-Data Editing: Each iteration of the design cycle (Figure 4.2) starts with the formation or modification of the input data for the simulation. This data contains the circuit description, the analysis requests, and the control options for the simulation. Even though a program like the Expert Emulator is capable of modifying the input data automatically, direct access to the design database is essential to the designer and is accomplished using editors. In the past, *text editors* were the only type of editors available for this task. They are still widely used today. Designers using text editors need to know the exact input-language syntax for the simulators they use. Syntactic errors are common when text editors are used.

Schematic editors are a big improvement over text editors, since they let designers edit the schematic of the circuit. Diagrams (on paper) are used even when only text editors are available. However, the task of translating the circuit description from the schematic to the textual format is done automatically by the schematic editor, thereby eliminating most syntactic errors and saving time. Schematic editors usually simplify the entering of analysis requests and simulation options with a friendly interface. The

visual text editors Vi and Emacs are currently available, but schematic editors, such as OctSPICE [Laidig90] and iSPI [Acuna90], could be included as well.

Simulation-Error Recovery: The Expert Emulator, introduced in Chapter 2, can be invoked by the designer whenever simulation errors occur. Modifications to the input data are not only suggested but made automatically. The user, however, can override the modifications.

Pre-Simulation Data Checking: Errors caught early cost less, since in this way unnecessary and possibly CPU-expensive simulation runs can be avoided. Therefore, it may be beneficial to run the Expert Emulator before a simulation, using an appropriate set of rules. The program SPLINT [Kuo90] is a SPICE-format checker that may also be used for this purpose. SPLINT is a conventional procedural program and can not be augmented with new rules as easily as the Expert Emulator.

Circuit-Design Aid: The Expert Emulator has a third use, described in Section 3.6. It can be used to criticize a design and to suggest improvements based on results and formulas derived from circuit analysis. A similar technique has been presented in [Spickelmier89]. For this task, the design rule set in the knowledge base is used.

4.4 Data Flow and Organization

The flow of data between the various tools in NECTAR is illustrated in Figure 4.4. The user accesses NECTAR through a computer terminal (TE). A circuit editor (ED) can be used for the initial formulation or for modifications of the circuit and analysis (input) data (ID). Input data is sent to a simulation program (SI) running on one of the machines (MA) available in the computing environment (CE). Simulation results are stored in the output database (OD) and displayed on the terminal by a post-processor (PP). When invoked, the inference engine (IE) of the Expert Emulator (EE) executes rules from the knowledge base (KB) driven by the problem data from the input and output databases. The user can access the knowledge base to add or modify rules with the rule editor (RE). Through the user interface (UI) the user controls the framework. The user-interface module and the Expert Emulator can alter the status variables of the framework (FS), which determine the

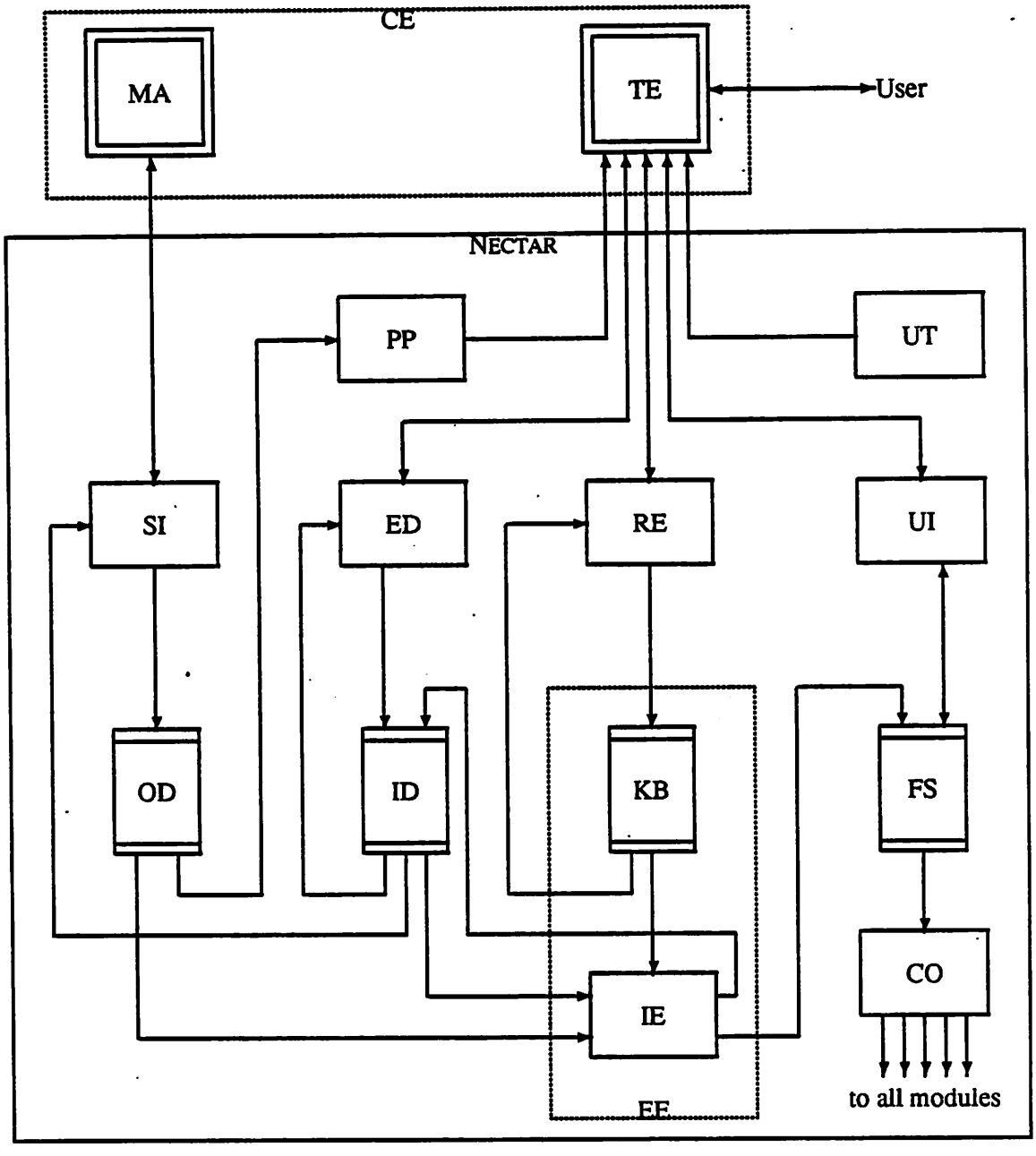


Figure 4.4: Framework data flow between simulators (SI), post-processors (PP), editors (ED), the Expert Emulator (EE) [composed of an inference engine (IE) and a knowledge base (KB)], the Rule Editor (RE), the user interface (UI), and utility programs (UT) running in a distributed computing environment (CE) with various machines (MA) and terminals (TE). ID and OD represent the simulation input and output data, CO is the framework controller, and FS is the framework status.

sequence of events through the framework controller (CO). Finally, various utility programs are available to send information to the user.

NECTAR stores data in the database as files. As a result, the communication between programs is simplified. More significantly, almost any program can be included in the framework, since programs are not restricted to conform to a specific data model. Only the more common formats are used for storage. The predominant formats are those of SPICE input and SPICE output. Files are named automatically by NECTAR and can be accessed using framework utility functions. Other utilities allow directory browsing and can recognize files containing data in certain formats.

Formats required by less frequently used programs typically are not present in the database. Special database-to-program translators convert data from the available formats. Similarly, output from those programs is converted with program-to-database translators before stored in the databases. Figure 4.5 illustrates the use of data-format translators for the Expert Emulator. The DB-to-EE translator converts data from SPICE input and output

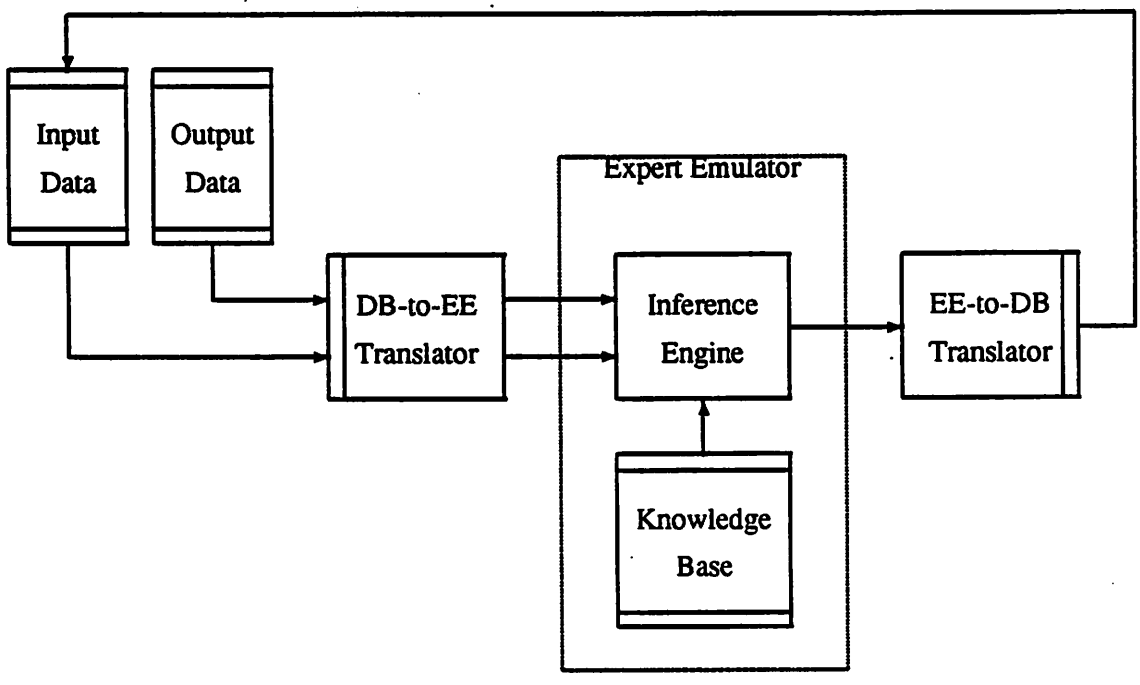


Figure 4.5: Data-format translators for the Expert Emulator

formats and the EE-to-DB translator converts data to SPICE input format.

Even for the frequently used SPICE2, the "raw" part of the output is first converted to SPICE3 "rawfile" format and then stored. The reason for this conversion is that the more likely use of the "raw" data is plotting it using Nutmeg, the post-processor of SPICE3. The translation is done with the program Sconvert, as illustrated in Figure 4.6.

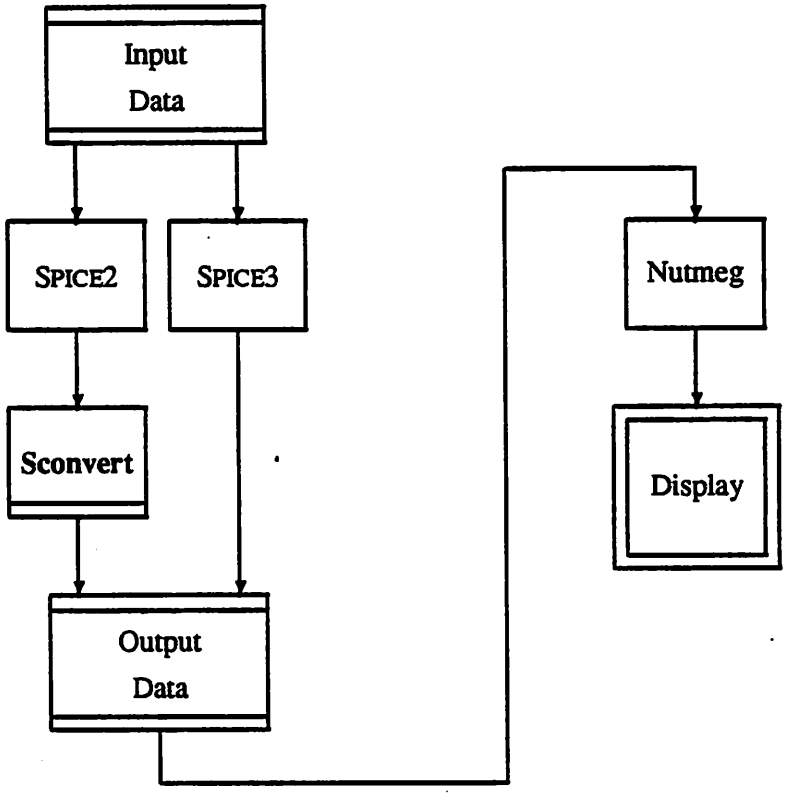


Figure 4.6: SPICE3 post-processor Nutmeg plots results from SPICE2 translated by the program Sconvert (SPICE2-to-DB translator)

4.5 Integration in a Diverse Computing Environment

A typical computing environment for IC design today includes several, possibly different, machines interconnected in a network. NECTAR accommodates and takes advantage of distributed and heterogeneous computing environments. Integration of computing environments has been simplified owing to several advances in computer networking.

CPU-intensive simulation jobs can be sent from NECTAR for execution on remote machines that are either more powerful or less loaded. Operating-system utilities are used for the communication between machines. The communication across UNIX machines is based on the remote shell command `rsh`, which connects to the remote machine and executes the specified command. Output data is passed back to the local host. For the communication between a UNIX host and a remote IBM 3090 machine, the `rje` utility program is used. This program is based on TCP/IP mail. In either case, the user of NECTAR does not need to know how the communication, which remains hidden, is achieved.

Multiple processes may run in parallel in a multitasking operating system. Running processes can be monitored and terminated with simple NECTAR commands, which use operating-system utilities, such as `ps` and `kill`.

Design and other data on a remote file system can be accessed from within NECTAR. Remote access is based on the Network File System (NFS), a file system implementation that allows sharing of ordinary files and directories in a multivendor networking environment.

4.6 The Flow of Control in the Framework

NECTAR is invoked from the UNIX shell with the command "`nectar`". By entering NECTAR, users enter a command loop that continues until the "`quit`" command is issued. Using the task commands, one can invoke tools without having to remember the syntactic details of how each tool is invoked on its own. More details on the user interface are given in Chapter 5. Other commands serve to set the various framework status variables and options, such as the execution machine, file names, the pager and editor of choice, and others.

Of particular interest is the flow of actions that correspond to the edit-simulate-examine design loop shown in Figure 4.2. A framework-control cycle for this design loop has been incorporated in NECTAR. The corresponding flowchart is shown in Figure 4.7.

First, the circuit is edited with a circuit editor. Then, if desired, the presimulation rules of the Expert Emulator are applied to find problems with the simulation input data.

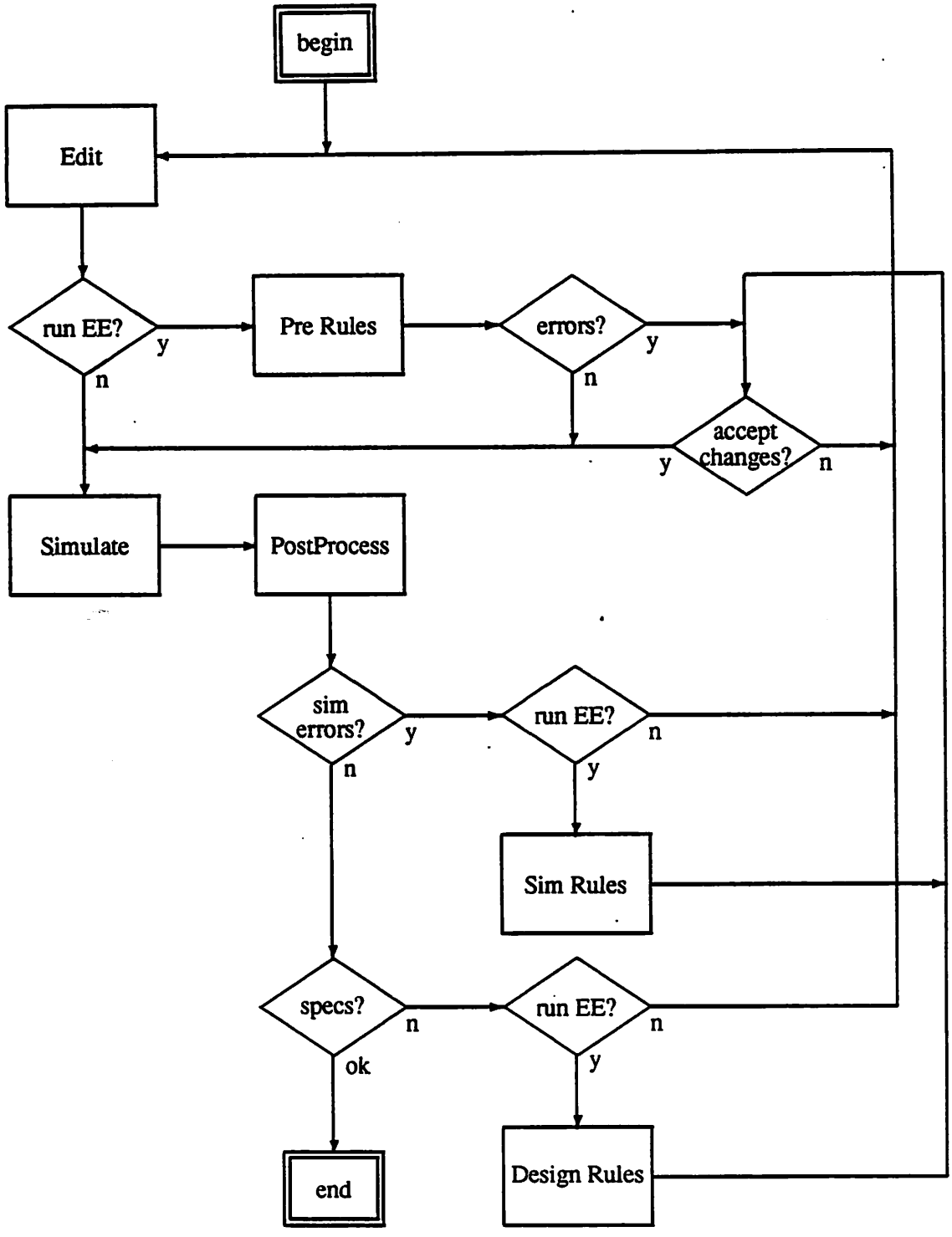


Figure 4.7: NECTAR flowchart corresponding to principal analog design cycle

If no errors are found or errors are found and the corrections made are acceptable, the process proceeds to the simulation phase. If errors are found but the corrections made are not acceptable, the circuit editor is invoked again.

In the simulation phase, after the simulation run is over, a post-processor displays the simulation results. If simulation errors occurred, the simulation-error rules of the Expert Emulator can be applied. If the Expert Emulator comes up with acceptable corrections to the input data, a new simulation is started. If no rules are relevant or the corrections made are not acceptable, the process goes back to the editing phase.

Once the simulation concludes with no errors, the simulation results are checked against the design specifications. If the specifications are met, the design is over. Else, the design rules of the Expert Emulator can be applied. As in the previous phases, acceptable corrections lead to a new simulation, whereas otherwise the editor is invoked.

Chapter 5

Improving the User Interface

5.1 Overview

The NECTAR framework is an interactive software package. This chapter presents issues and decisions on the design of the human-computer interface of NECTAR. Implementation aspects of the interface design are described in Chapter 6.

With the increasing use of computers, human factors in the design of software are receiving the due attention. Numerous design principles and guidelines have been obtained from practical experience and empirical studies. In one view of user interfaces, the user-computer "communication" is analyzed as a component of the general task fulfillment. Another analysis decomposes user interfaces into several increasingly refined levels, from conceptual to physical. The choice of an interaction style and the accommodation of novices as well as of experienced users are among the central design issues.

A task analysis at the conceptual and computer-system levels reveals the necessary functionality for the interface of NECTAR. The tasks have been mapped to two prevailing hardware platforms, the alphanumeric and the bit-mapped display. The two interfaces are compatible and can be used interchangeably during a project. In accordance with principles of "friendly" design, data entry is minimized, routine actions are automated, command arguments can be specified in several ways, and commands have a common "feel". Data is displayed in graphical form, whenever possible, and messages inform the user of system actions. The alphanumeric interface is based on a flexible command language, whereas the

bit-mapped interface uses multiple windows and a menu selection system.

A rule editor, a special interface to the knowledge base of the Expert Emulator, allows NECTAR users to create and modify rules while circumventing most of the syntactic details of the rules.

5.2 Issues in Computer-User Interfaces

This section presents analysis methods and design issues in user interfaces of interactive computer programs.

Often in the past, software designers would devote little time in the part of the program that handled the communication from and to the program user, the user interface. A software project would be considered finished when the core algorithms were developed and debugged. With the increasing use of computers by a larger and more varied pool of people, software developers have recognized the importance of user interfaces. "Human engineering is now understood to be the steel frame on which the structure is built" [Shneiderman87].

Designing "friendly" programs is not a simple matter. The field of human factors studies the implications of human characteristics on the design of equipment to be used by people. Some features of human-machine interaction, especially those specific to computers, are not well understood; hence, general principles of interface design are not yet sufficiently well developed. Instead, design guidelines have been compiled from experimentation, informal observation, and intuition. In [Smith82], hundreds of issues on functional capabilities, data entry, data display, and sequence control are covered through thorough lists of guidelines and checks. [Heckel84] specifies thirty "elements of friendly software design" that provide a variety of perspectives on interface design. Current good practice is based on guided evolution, an iterative design approach that guarantees flexibility by intentionally leaving some options open during the early stages of software development [Nickerson90].

The interaction of humans and computers can be analyzed from several points of view. Figure 5.1 illustrates the central concepts of the interaction from the task perspective

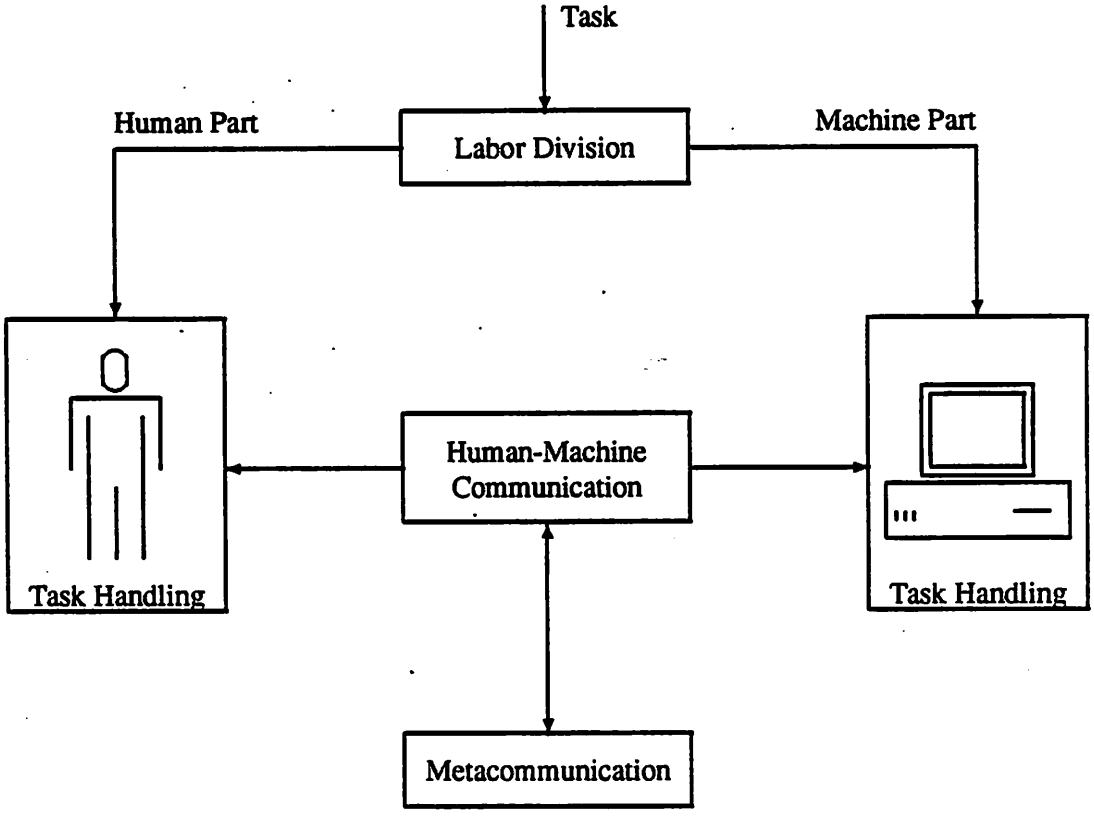


Figure 5.1: Central concepts in a task perspective of human-machine interaction

[Wærn89]. The general task — verifying or designing an analog circuit, in the context of NECTAR — is divided in two parts, one to be carried out by the human and one for the machine. The larger the machine part, the more automated the general process. In NECTAR, the input of the IC designer (human) is essential in each cycle of the design task, as presented in Chapter 4. The machine subtasks are computational, administrative, and diagnostic. In general, humans excel in creative, intuitive, and low-precision tasks, whereas computers are better at repetitive, algorithmic, high-precision, and routine tasks.

The human and the computer handle their respective tasks according to "task models" implicitly or explicitly present in the human's expertise and in computer programs, respectively. During task handling, commands, responses, data, and other information flows between human and computer. This "communication" is inherently "unnatural," because of the radical difference in levels of "intelligence." A well-designed interface narrows the "intelligence" gap by supporting many of the implicit assumptions that characterize the communication between humans. It is claimed that writing "friendly" software is an art involving techniques of effective communication [Heckel84].

A portion of the human-machine interaction consists of supplementary communication about the rules and requirements of the communication that is directly related to task handling. This supplementary communication is called *metacommunication* and includes help sessions, error messages, on-line tutorials, and information and commands on the system status. Since metacommunication intervenes in the main task, designers have to consider whether it should happen "actively" (for guidance) or on request, as expert users generally prefer.

A different analysis of user interfaces is presented in [Moran81]. This analysis introduces a hierarchical representation of command language systems, called the Command Language Grammar, that spans the conceptual (tasks and abstract concepts), communicational (command language, dialogues), and physical (I/O devices) aspects of user interfaces. The representation is made up of six description *levels*, illustrated in Figure 5.2, each level being a refinement of the previous levels.

At the Task Level, user needs are described in a way amenable to an interactive system. Concepts used by the system for the accomplishment of the tasks are introduced at the Semantic Level. Commands, arguments, state variables, and other syntactic elements

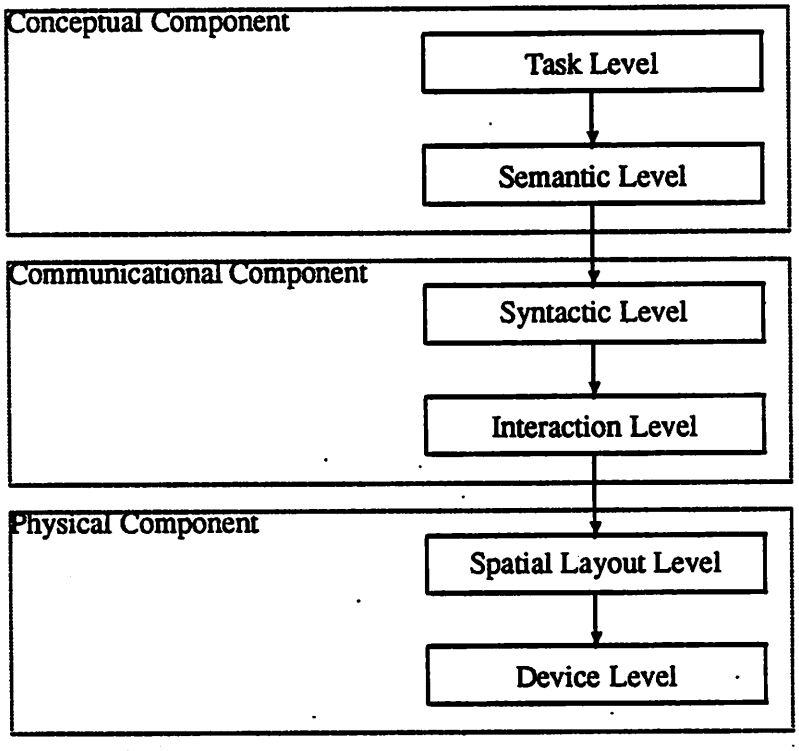


Figure 5.2: Level structure of the Command Language Grammar

are laid out at the Syntactic Level. The Interaction Level specifies the physical actions, such as key presses and mouse manipulations by the user and display actions by the system. The Spatial Layout Level describes the arrangement of the I/O devices and the display graphics. Finally, all remaining physical features are described at the Device Level. The description of each level contains procedures for accomplishing the tasks in terms of the conceptual entities and actions available at that level. The stratification into levels is not always precise. The needs of NECTAR call for a less abstract representation with fewer levels, as presented in Section 5.3.

Different styles can be chosen to support the user-computer interaction. The styles most in use are the following [Shneiderman87]:

- Menu Selection: The user chooses from a list of options presented by the system.
- Form Fill-In: Data is entered by the user in a particular format that simplifies repetitive actions.
- Command Language: Instructions are expressed directly using a language of a par-

ticular syntax.

- Natural Language: The communication is comprised of sentences from a restricted vocabulary resembling natural language.
- Direct Manipulation: The user moves and transforms objects on the screen directly, as if they were real objects; also described as "what you see is what you get" (WYSIWYG) interaction.

No one style is "best" in general, but one style may be better than the rest for a particular task. Menu selection and command language are the main contrasting styles, representing a trade-off between simplicity and power. Direct manipulation generally requires advanced hardware and sophisticated software and is being increasingly employed in all kinds of applications. It should be noted that the distinctions between different styles are sometimes blurred and that hybrid styles are common. Numerous rules and guidelines, applicable to one, several, or all interaction styles have been developed. Response time, system messages, screen design, and use of color are some additional important issues.

Among the challenges in user interface design is accommodating humans of varying profiles. Even though the users of a program such as NECTAR are much less diverse than, for example, the users of automated-teller machines, the general distinction into novice and experienced users still applies. Typically, novices are characterized by limited syntactic and semantic knowledge of the task and by anxiety about interacting with a computer. Experienced users have good knowledge of the task and familiarity with the computer. Knowledgeable but intermittent users are able to maintain the semantic knowledge of the task and the computer concepts but have difficulty retaining the syntactic knowledge. Each class of users has different needs about guidance, speed, and feedback. Novices prefer selecting to giving commands, feel more confident with full terminology, and require generous prompting, error messages, and on-line assistance. For expert users, it is important to be able to work rapidly, to avoid being disturbed by extensive messages, and to be equipped with shortcuts, abbreviations, and macros for frequent actions. Intermittent users need, at least in the beginning, novice-like interaction to refresh their memory.

Several CAD programs have addressed user interface issues. VEM [Harrison89], an interactive graphics program for the Berkeley Design Environment, uses multiple over-

lapping windows, deck-of-cards pop-up menus, dialogs, and other user interface entities. The user of VEM can specify commands using the menus, using single keystrokes, or by typing the command name. Cleopatra [Samad86] is a natural language interface for circuit-simulation post-processing. An evaluation of Cleopatra has led to the conclusion that a graphics interface would be preferable to natural language for the CAD domain [Cobourn87]. ECSTACY [Shyu88], an interactive IC optimization system, uses a form fill-in interface for the detailing of problem specifications. The Cadweld CAD framework [Daniell89] provides an icon-driven interface comprised of several views that represent tool classes and hierarchies and the flow of control and data.

5.3 Designing The User Interface of NECTAR

5.3.1 Assigning Representation Levels

The six-level structure of Figure 5.2 can be simplified for the needs of NECTAR. The merging of the Semantic and Syntactic Levels into a single level, the System Level, and, likewise, of the Spatial Layout and Device Levels into the Physical Level, results into a more concise four-level structure, as illustrated in Figure 5.3. The Task and Interaction Levels are identical to the corresponding levels of the Command Language Grammar (Section 5.2). The System Level represents the system operations that are necessary to accomplish the tasks, as well as how the operations are evoked. The Physical Level describes the physical arrangements and features of the I/O devices.

An example may help make some of the notions of the level structure more concrete. At the Task Level, one might want to "simulate a circuit with SPICE". The entities at this level are: circuit, simulation, and simulation type. The same action is represented at the System Level with the notion of "invoking the `spice` executable with input the file containing the SPICE description of the circuit". The entities at this level are: command, executable, arguments, and system files. At the Interaction Level, the action is represented by the keystrokes of the user at the system prompt (for a UNIX system: "`spice < input-file`".) Finally, at the Physical Level, the action is described with additional information about the screen layout, the relative positions of the system prompt and the

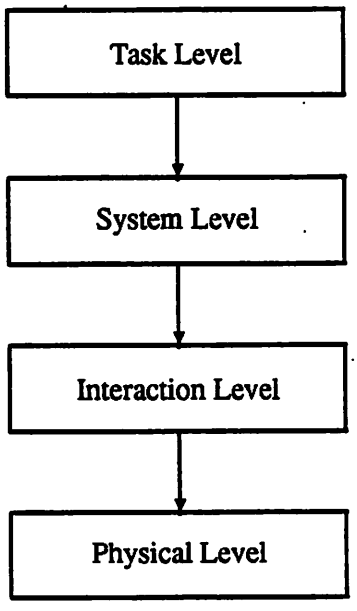


Figure 5.3: Level structure for the user interface of NECTAR

"echo" of the user's command, fonts, colors, and other screen details.

5.3.2 Task Analysis

Essential in the design of the user interface of NECTAR is to identify the necessary functionality, i.e., the tasks that must be accomplished. The functionality of the interface must be adequate but not excessive, because unnecessary complexity may confuse and discourage the users.

Necessary tasks are the design tasks supported by the framework, as listed in Section 4.3: various types of simulation, post-processing, design editing, simulation-error recovery, data checking, and design aid. In addition, the analog-design cycle, as represented by the flowchart of Figure 4.7 in Chapter 4, must be supported as a compound task. A help facility, a history mechanism, and a bug-report feature are tasks that fall in the category of metacommunication.

In addition to the tasks above, which represent decisions at the Task Level, several new tasks are introduced at the System Level. These tasks relate to system entities not represented at the previous level, such as files, directories, commands, paths, computer

jobs, and state variables.

The various types of tasks are summarized in Table 5.1. Tasks introduced at the Task Level are represented at the System Level as well, but not vice versa. Even though tasks introduced at the System Level do not directly promote the principal goal of design verification, they do represent useful utilities.

Interface Level of Task Introduction	Task Type	Task
Task Level	Simple	Simulation
		Post-processing
		Editing
		Error recovery
		Data checking
		Design aid
	Compound	Analog-design cycle
	Metacommunication	Help
		History
		Bug report
System Level	File system	File-name setting
		File deletion
		Directory listing
		Directory change
	Computer jobs	Job monitoring
		Job termination
		Parallel-job control
	State variables	Status check
		Status change

Table 5.1: Summary of tasks for the user interface

5.3.3 Choices at the Interaction Level

In choosing an interaction style, one first needs to consider the type of hardware that the program will be running on, and in particular the type of computer terminal that will be used. Engineering workstations with bit-mapped displays are widely used among IC designers, but alphanumeric terminals used with minicomputers and mainframes are also in use, at the work place and remotely through low-bandwidth lines. Mixed use of both types of terminals is not unusual.

Although bit-mapped terminals generally allow the design of better interfaces than do alphanumeric terminals, either type would be adequate for most NECTAR tasks. Targeting only one type of terminals would be an unnecessary restriction to potential users. Hence, two interfaces have been developed for NECTAR, one for alphanumeric and one for bit-mapped displays.

The two interfaces are identical at the Task and System Levels and, thus, have the same functionality. They differ at the Interaction and Physical Levels in order to exploit the different capabilities offered by the hardware. Detailed descriptions and design choices at the Interaction and Physical Levels are presented in Section 5.5 for alphanumeric displays and in Section 5.6 for bit-mapped displays.

Despite their differences, the two interfaces are compatible. Work started using one interface can be continued using the other interface with no need for data adjustments and transforms. This is possible because NECTAR stores data in the databases as files, as mentioned in Section 4.4, and also because of the equivalence of the two interfaces at the System Level.

5.4 Data Entry and Display

This section presents principles of interface design that have been applied to the two NECTAR interfaces.

A guideline for data entry is the minimization of input actions by the user. In NECTAR, users's actions are minimized in several ways. Most choices are made using one or two keystrokes or a mouse click. Simple framework commands have substituted lengthy

command sequences at the operating-system level. This reduces training and memorization requirements. As an example, the UNIX command

`spice`[1] `-r` *rawfile* `<` *infile* `>` *outfile*

is substituted with the keystroke "2" or a click of the mouse on the "SPICE2" menu entry (assuming that the name of the input file has been specified in advance).

Certain commands, when invoked from the operating system, require certain environment variables or auxiliary files — such as "dot" files in UNIX systems — to be in a certain state. Whenever changes to ensure such requirements can be made without the user's intervention, actions are taken and remain "hidden" from the user. As an example, the `rje` UNIX program, used for the communication with IBM mainframes, does not work properly if the user's auxiliary mail file, `.mailrc`, contains a "`set record=...`" entry, which allows outgoing mail to be saved in a file. When the above condition is present, NECTAR creates a temporary `.mailrc` with no entry for recording and then executes the `rje` command. The `.mailrc` file is restored to its original form after the task is completed.

Redundant data entry is avoided in NECTAR, thus preventing user annoyance and the possibility for errors. Any required argument of the stand-alone command for a tool may be omitted when the tool is invoked inside the framework, if that argument can be derived from the context of previous commands during a session with NECTAR.

Names for files used to store results from various tools are given default names by the framework. The user can access those files using framework commands without having to know the names. A user that insists on specifying the file names can do so using the appropriate NECTAR commands.

Arguments for framework commands can be entered in several ways: some can be included as options with the NECTAR-invocation command; all arguments can be given as regular supplements together with the commands; finally, if not supplied previously, arguments are inquired with questions to the user. This flexibility gives the user additional control on data entry.

Actions and subtasks most likely needed after the completion of a user-requested task are invoked automatically. As an example, after a simulation, the default post-processor

[1] "`spice`" refers to SPICE2.

is automatically invoked to display the simulation results. As another example, modifications to the input file by the Expert Emulator are not only suggested but also made automatically. The user can always override such actions.

NECTAR takes advantage of initialization files for interactive programs (`.init` files, in UNIX) to automate actions that would otherwise require typing. As an example, NECTAR copies (with appropriate modifications) plotting or printing commands from in the SPICE input file to file `.spiceinit`, the initialization file for SPICE3 and Nutmeg. In this way, the desired plotting command is executed automatically. As another example, the use of the initialization file compensates for the lack of a batch running mode in the steady-state simulation program SSPICE [Ashar89]. Instead of requiring the typing of the command "`steady ...`" for each simulation run, NECTAR executes the respective commented-out line from the input file using `.spiceinit`.

Framework commands are characterized by a common "feel". Consistency within the command language is more helpful to a user than compatibility between the command language and the natural language [Wærn89]. The interface shields the user from command differences across different machines.

Communication to the user (data display) is made in graphical form, where appropriate. Graphs, dialog boxes, and visual metaphors relieve the need to read and interpret alphanumeric data.

NECTAR reports to the user all actions taken with system messages. In particular, modifications by the Expert Emulator are reported together with justifications for the changes. If a rule is executed with a low degree of certainty, the user is asked before any modifications are made whether the proposed changes are acceptable.

Computer response time significantly affects the user's productivity and level of satisfaction. Care has been taken in implementing the user interface of NECTAR to ensure short response times — less than a second is considered adequate [Shneiderman87]. Lengthy jobs, such as some simulation runs, are handled either by executing them in the background or by continually displaying the elapsed CPU time.

5.5 The Interface for Alphanumeric Displays

A flexible command language combined with user queries has been designed for the alphanumeric interface of NECTAR. Figure 5.4 lists the available commands, as they

```
nectar 1> help

nectar commands

2  : spice2              3  : spice3           3b : spice3b1+
3c : spice3c1            3d : spice3d1         bb : jobs on bigboote
bg : jobs in background  br : report a bug     bs : spice3 w/ bsim
cd : change directory    cl : clean files      co : codecs
cs : spice2 w/ cosines   d  : subdirectories   fg : jobs in foreground
fm : spice2 w/ new fm    fs : fort spice2      h  : help
hi : history             i  : new input file   ia : spice3 interactive
ib : submit to the IBM   k  : kill spice jobs  l  : list input files
le : view input file     n  : nutmeg           o  : view output file
q  : quit                r  : remote machine   sd : set display
se : set editor          so : save output      sp : set pager
sr : save raw output     st : status           ss : sspice
t  : spice cpu time      v  : edit input file  ve : spice version
x  : expert emulator     xs : xsplot
```

Figure 5.4: A listing of NECTAR commands taken from the `help` facility

are displayed by the help facility of the framework.

To serve both the experienced and novice users, commands may be entered with full names or using abbreviations. All non-ambiguous abbreviations of command names are legal. To comply with the outcome of relevant studies that have shown two-letter abbreviations to be optimal, all commands can be shortened to two letters. Some of the more common commands hold one-letter abbreviations as well.

The command style is influenced by UNIX shell commands. The framework system prompt includes the name of the program and the number of the command.

5.6 The Windowed Interface

A *window* is a portion of a display screen, typically assigned to a group of information or to one of several simultaneously running processes. Modern engineering

workstations with high-resolution bit-mapped graphics displays make multiple windows practical. Windows have transformed human-computer communication from linear flow of information to multi-dimensional parallel interaction. Many advances, such as overlapping windows, pop-up menus, multiple fonts, direct manipulation using the mouse, and the desk-top paradigm have been introduced with the Xerox Star [Johnson89] and popularized with the Apple Lisa [Williams83] and Macintosh [Apple85] and recently with the Microsoft Windows 3.0 [Udell90].

The NECTAR interface for bit-mapped displays uses several windows and is menu-driven. Figure 5.5 shows the screen layout of the interface. The main windows are tiled, whereas auxiliary windows are overlapping. Two text windows are used for the display of input data (the larger one on the top) and messages (the one at the bottom). They are both scrollable with left-hand-side scrollbars. The input-data window is editable and used to modify directly the input data.

A menu between the two text windows has entries ("buttons") for the framework commands. A single static menu has been chosen because the number of tasks is relatively small. To avoid confusion, the mouse buttons are equivalent: any button may be pressed with the identical result. Menu actions are confirmed with the use of reverse video.

Pop-up dialog boxes are used for argument entry and command confirmation. The visual metaphor of a thermometer-like linear ruler displays the elapsed time of CPU-intensive jobs as they run. This helps eliminate the "Is the computer down?" syndrome [Shneiderman87] of computer users, who are anxious about the state of their jobs.

Color groups objects on the screen in a consistent way and creates an aesthetically pleasing image. Only a few colors are used, since excessive use of colors disorients the user.

5.7 A Mechanism for the Easy Addition of New Rules

As mentioned in Chapter 2, knowledge-based systems can be easily expanded with the addition of new rules to the knowledge base, which is separate from the inference engine. In NECTAR, the expandability of the Expert Emulator is further enhanced by a special rule editor developed to ease the process of composing new rules. By using the rule

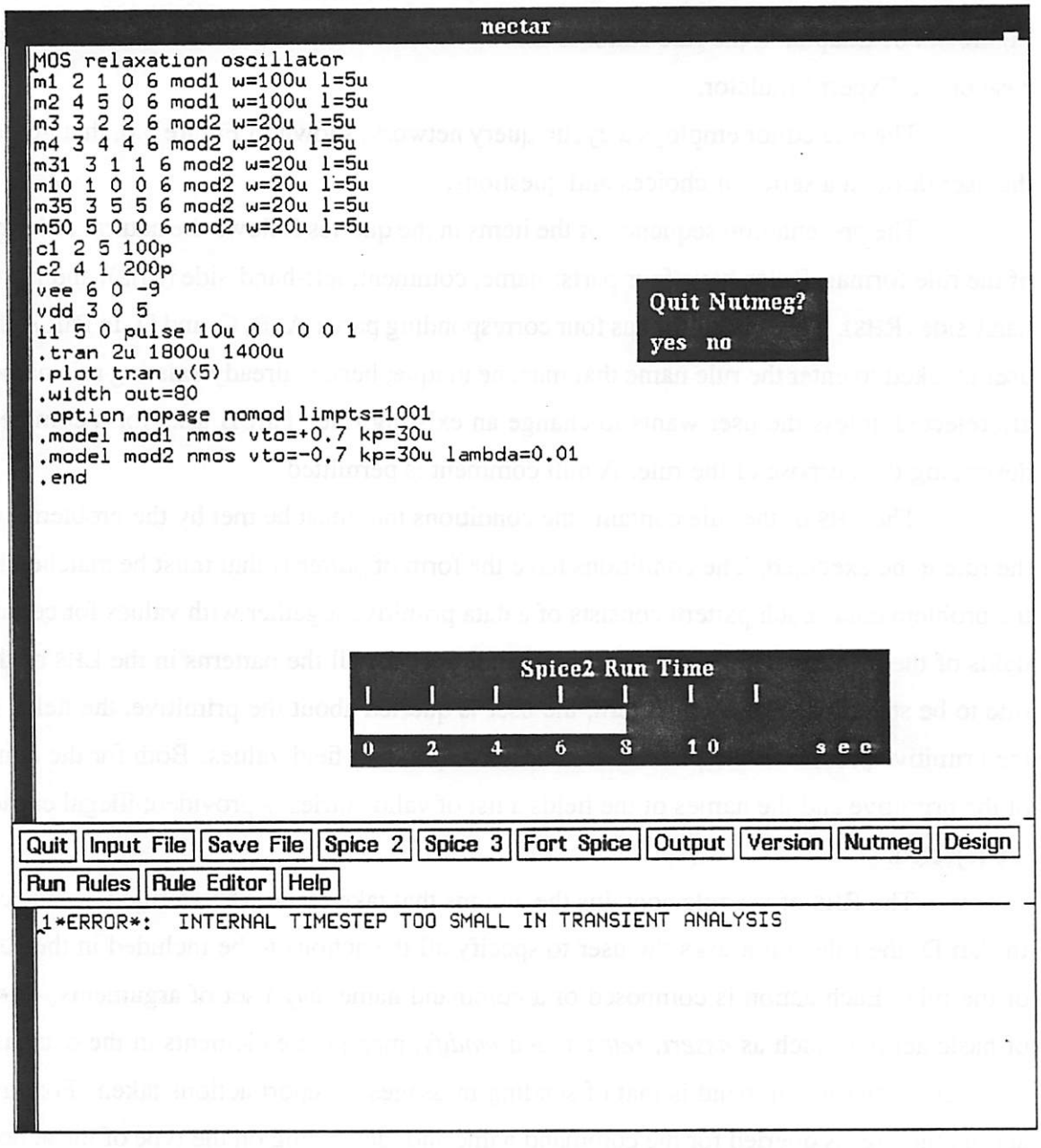


Figure 5.5: The windowed user interface for NECTAR

editor, NECTAR users can bypass most of the syntactic details of the rules. As shown in Figure 4.4 of Chapter 4, the rule editor is the interface between the user and the knowledge base of the Expert Emulator.

The rule editor employs a cyclic query network, shown in Figure 5.6, that guides the user through a series of choices and questions.

The presentation sequence of the items in the queries follows the natural ordering of the rule format. Rules have four parts: name, comment, left-hand-side (LHS), and right-hand-side (RHS). The rule editor has four corresponding parts: A, B, C, and D. In Part A the user is asked to enter the rule name that must be unique; hence, already existing rule names are rejected unless the user wants to change an existing rule. Part B asks for a comment describing the purpose of the rule. A null comment is permitted.

The LHS of the rule contains the conditions that must be met by the problem, for the rule to be executed. The conditions have the form of *patterns* that must be matched by the problem data. Each pattern consists of a data primitive together with values for certain fields of the primitive. In Part C, the rule editor asks for all the patterns in the LHS of the rule to be specified. For each pattern, the user is queried about the primitive, the fields of the primitive present in the pattern, and the corresponding field values. Both for the name of the primitive and the names of the fields a list of valid entries is provided; illegal entries are rejected.

The RHS of the rule contains the actions that take place when a rule is executed. In Part D, the rule editor asks the user to specify all the actions to be included in the RHS of the rule. Each action is composed of a command name and a set of arguments. A set of basic actions, such as *assert, retract,* and *modify*, manipulate elements in the database. Another common command is that of sending messages to report actions taken. For each action, the user is queried for the command name and, depending on the type of the action, the primitive with its fields and values, the text of the message, or a list of other arguments. Again, illegal endries are rejected.

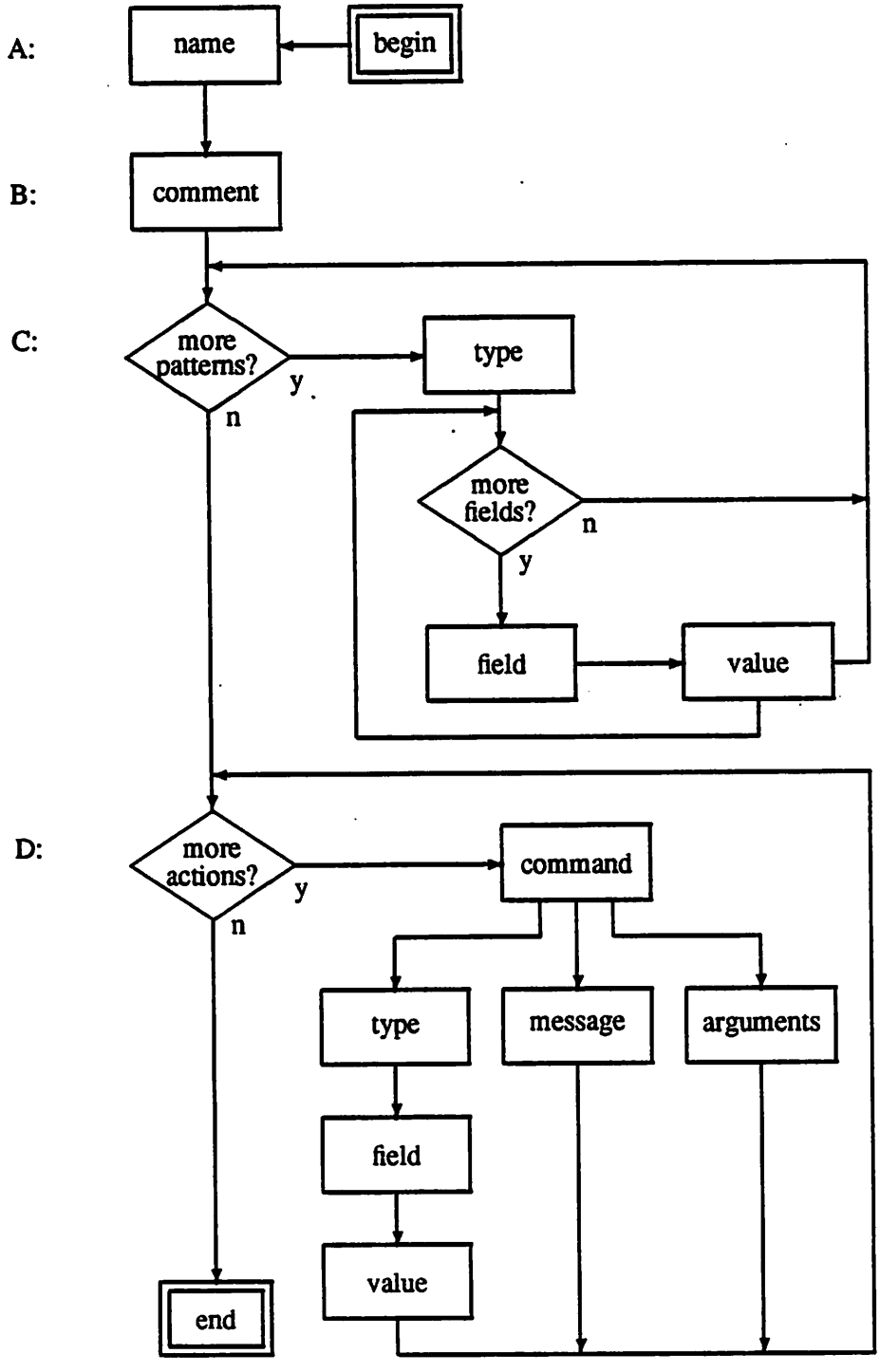


Figure 5.6: The query network for the Rule Editor

Example

To illustrate the use of the Rule Editor, a session with the editor, involving the creation of the rule "bjt-base-resistance," is described. The following is the complete text of the interaction using an alphanumeric display, with the user's input in sans-serif font.

A) Specify rule NAME : bjt-base-resistance
B) Specify rule COMMENT :
C) Next, for the 'IF' PART, specify the rule patterns.
Valid pattern types are: resistor capacitor nonlinear-capacitor inductor
nonlinear-inductor mutual-inductance lossless-transmission-line vccs
vcvs cccs ccvs voltage-source current-source bjt mos model analysis
option width error node
Are there more patterns? [ny]: y
C1) Specify the pattern type : bjt
Next, specify the pattern fields.
Valid fields are: name collector base emitter model
Are there more fields in this pattern? [ny]: y
Specify the field name : model
Specify the field value : var m
Are there more fields in this pattern? [ny]: n
Are there more patterns? [ny]: y
C2) Specify the pattern type : model
Next, specify the pattern fields.
Valid fields are: name type af beta bf br bv cbd cbs cgd cgs cgbo cgdo
cgso cj cjc cje cjo cjs cjsw delta eg eta fc gamma ibv ikf ikr irb
is isc ise itf js kappa kf kp lambda ld level m mj mjc mje mjs mjsw
n nc ne neff nf nfs nr nss nsub pb phi ptf rb rbm rc rd re rs rsh
tf theta tox tpg tr tt ucrit uexp uo utra vaf var vj vjc vje vjs
vmax vtf vto xcjc xj xqc xtb xtf xti
Are there more fields in this pattern? [ny]: y
Specify the field name : name
Specify the field value : var m
Are there more fields in this pattern? [ny]: y

Specify the field name : rb
Specify the field value : var r and 0 or nil
Are there more fields in this pattern? [ny]: n
Are there more patterns? [ny]: n
D) Next, for the 'THEN' PART, specify the actions.
Are there more actions? [ny]: y
D1) Specify command : print
Specify message : Base resistance of var m has zero value.
Are there more actions? [ny]: y
D2) Specify command : bind
Specify arguments : var new-value 100
Are there more actions? [ny]: y
D3) Specify command : modify
Specify object : pat2
Specify field name : rb
Specify new value : var new-values
Are there more actions? [ny]: y
D4) Specify command : print
Specify message : MODIFICATION: Base resistance of var m changed to var new-value ohms.
Are there more actions? [ny]: n
End of session.

As a result of this session with the Rule Editor, the following CLIPS rule is created and added to the knowledge base:

```
( defrule bjt-base-resistance ""
  ?pat1 <- (bjt (model ?m))
  ?pat2 <- (model (name ?m) (rb ?r&0|nil))
  =>
  ( fprintout inf-file "Base resistance of " ?m " has zero value." t)
  ( bind ?new-value 100)
  ( modify ?pat2 (rb ?new-values))
  ( fprintout inf-file "MODIFICATION: Base resistance of " ?m
```

```
        " changed to " ?new-value " ohms." t)
    )
```

Chapter 6

Implementation

6.1 Overview

The various novel concepts introduced in the previous chapters have been implemented as computer programs. These programs have served as test vehicles for the evolution of ideas and have shown the feasibility of the proposed solutions. This chapter describes the choices made in implementing the programs and the experience with the programming platforms used.

In implementing the Expert Emulator, the use of an efficient algorithm for the inference engine of the production system is of primary importance. The Rete Match Algorithm, described in Section 6.2.1, optimizes several aspects of the pattern-matching process, the most computationally intensive function of the inference engine. The reasoning strategy that best suits searching in the problem space of the Expert Emulator is forward chaining. Since the Rete algorithm is an efficient pattern matcher and favors forward chaining, it has been adopted for the Expert Emulator.

A first prototype of the Expert Emulator was implemented in Common LISP using an unoptimized but flexible locally developed inference engine. While this version established the validity of the Expert-Emulation Approach, it was hampered by slow execution and large memory requirements. Consequently, a second version has been developed using CLIPS, a C-based public-domain expert-system tool. While maintaining the functionality of the first version, the CLIPS version has reduced both the execution time and memory needs

by over one order of magnitude.

The NECTAR framework also has two implementations, one for each version of the user interface. The alphanumeric version has been implemented using UNIX shell scripts. Although this version was initially intended to serve only as a prototype, it has acceptable run-time performance and has been retained for its simplicity and direct access of UNIX functions. The windowed version has been implemented in C, using the X Window System (X). Many low-level programming details of the X protocol have been handled automatically with the X Toolkit, a library package layered on top of X.

6.2 Implementation of the Expert Emulator

The choice of the production-system programming model for the Expert Emulator (Section 2.5) is essential for the organization of the domain expertise (in rules). Although the notion of dividing the domain knowledge into rules might be considered an implementation issue, it is central to the Expert-Emulation Approach and, hence, presented together with the principles of the approach in Chapter 2 and detailed further in Chapter 3. The user of the Expert Emulator should be aware of the rule structure, particularly in view of the "incompleteness" of the rule set, as explained in Section 3.3. This section presents the less visible implementation aspects of the Expert Emulator, including the choices of a pattern-matching algorithm, a reasoning strategy, and a programming language.

6.2.1 Algorithmic Considerations

Production systems were originally conceived to formalize symbolic logic questions; they were shown capable of representing general problem-solving knowledge [Post43]. However, the theoretical foundation was not thorough enough to produce practical programs. Algorithms were needed to accomplish the tasks of an inference engine (Section 2.5.1), in particular pattern matching and conflict-set resolution (rule selection).

The first suggested rule control strategy was based on assigning a static priority order to rules [Markov54]. As the use of production systems spread, it was seen that systems built straightforwardly according to the definition are computationally expensive. An

unoptimized pattern matcher would sequence through the rules in a certain order, comparing each rule against all data elements, until all conditions in the "if"-part of a rule were satisfied. This process would be repeated for each cycle of the inference engine (Figure 2.4). It has been shown that significant improvements in efficiency can be gained by minimizing attempted matches of condition elements in unsatisfied rules with unrelated data elements [McDermott78]. Such studies resulted in an algorithm that greatly improves efficiency, the *Rete Match Algorithm* [Forgy82].

The Rete algorithm eliminates most of the redundant work that a straightforward unoptimized pattern-matching algorithm would do. Since most of the data remains unchanged after each inference cycle — this property is called *temporal redundancy* —, the bulk of the attempted matches of an unoptimized algorithm are identical across cycles [Brownston85]. The Rete algorithm takes advantage of temporal redundancy by not recomputing all the matches on each cycle but looking only for changes in matches. As illustrated in Figure 6.1, changes in matches originate in changes in data and result in changes to the conflict set (Section 2.5.1), which is stored between cycles.

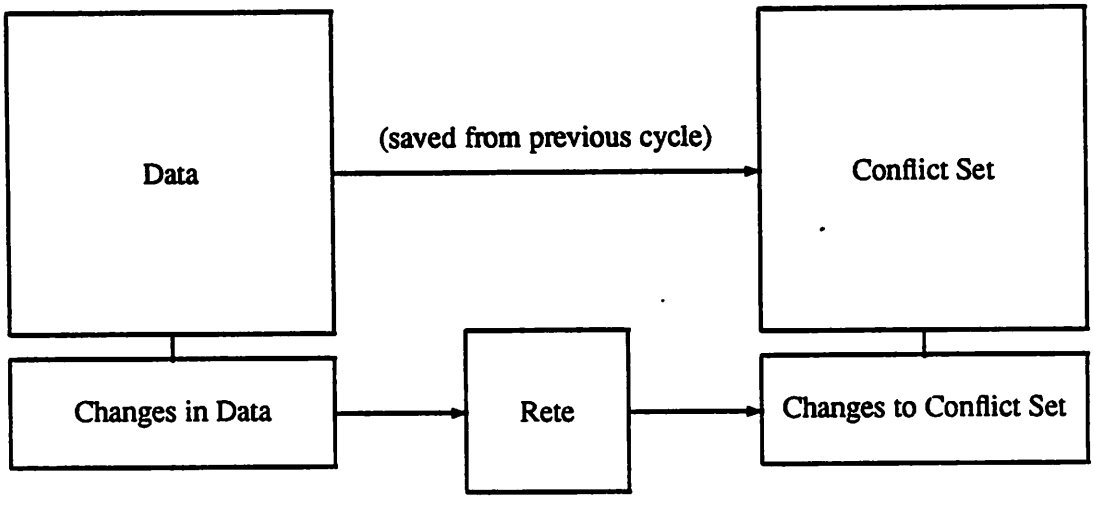


Figure 6.1: The incremental matching strategy of the Rete algorithm

The Rete algorithm reduces the dependence of the matching process on the number of rules by exploiting the *structural similarity* between rules. Conditions shared by multiple rules are evaluated only once. Another computationally expensive part of the matching process involves the calculation of whether a rule, whose conditions all match

individually, matches as a whole. This is not self-evident, since variables appearing in multiple condition elements of a single rule must be matched with identical data, for the rule to match as a whole. Instead of computing the cross products of all combinations of matches to all condition elements and do this on every cycle, the Rete algorithm stores partial combined results and uses them on later computations. Matches for individual conditions as well as partial combined results are stored in a tree-structured sorting network whose nodes represent the condition elements [Forgy82].

The *reasoning strategy* in a production system refers to the direction of the series of inferences that connect a problem to its solution. *Forward-chaining* systems start from known facts and proceed toward the solutions. By contrast, *backward-chaining* systems start from a hypothesis, break that up into intermediate hypotheses, and continue until known facts are reached. Forward-chaining systems are more appropriate when there are many acceptable solutions and the length of the inference chain is short, as shown in Figure 6.2a. Backward chaining is more appropriate when the search network is narrow and deep, as illustrated in Figure 6.2b [Giarratano89]. The distinction between the two strategies is not absolute. Many systems implement both reasoning strategies and systems with either strategy can be programmed to emulate the other one.

The search space of the Expert Emulator has many final states (solutions), namely the various corrective techniques. In addition, final states usually are arrived at after only a few inferences. As explained above, such characteristics (broad and not deep search networks) call for a forward-chaining reasoning scheme. Since the Rete algorithm is not only efficient but also favors forward chaining, it has been chosen as the pattern-matching algorithm for the Expert Emulator.

6.2.2 Using LISP

Several production-system languages, such as OPS5, OPS83, and ART, are based on the Rete algorithm. Among them, OPS5 [Forgy81], developed by the designer of the Rete algorithm and available in the public domain, has been the most widely used. OPS83 [Forgy84] and ART [Inference89] are two of the numerous commercial expert-system building tools available today at costs of thousands to tens of thousands of dollars. OPS83 is a

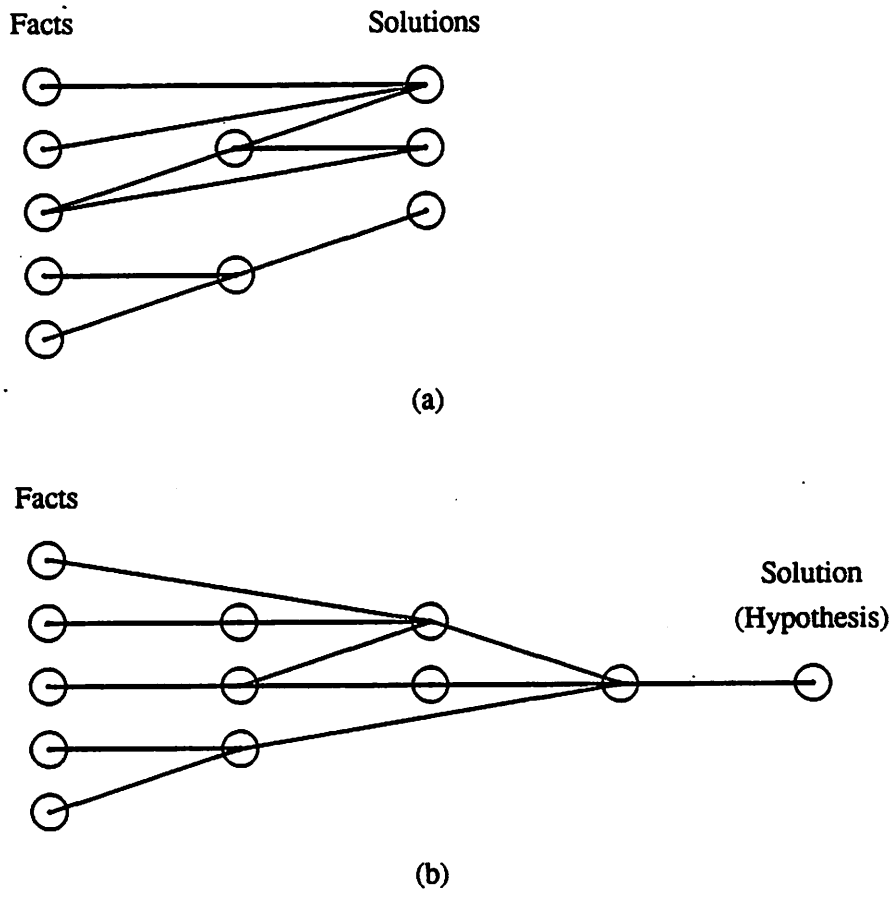


Figure 6.2: Appropriate applications for reasoning strategies: (a) forward chaining; (b) backward chaining

compiler-based descendant of the OPS family that combines a rule-based and a procedural programming paradigm, whereas ART is a multi-paradigm tool that supports forward and backward chaining, hypothetical reasoning, and object-oriented programming among other features.

During the planning stage of the implementation of the Expert Emulator, it was decided that the acquisition of a commercial tool was not justified. At the time, the principal goal had been the demonstration of the proposed approach through a quick prototype. The public-domain alternative, OPS5, presented two problems:

1. Restrictive data structures and expressions for the conditions in rules.
2. Non-standard underlying language: OPS5 is written in a particular dialect of LISP, Franz LISP [Foderaro83, Wilensky84]. Unlike many other languages, LISP has evolved to several dialects that differ considerably from one another. The lack of a language standard spurred a certain activity in the mid-1980's that resulted in the development of Common LISP [Steele84], a dialect intended to serve as a standard to which each implementation of LISP would make any necessary extensions. The clear trend toward commonality and portability had suggested that a LISP-based implementation of the Expert Emulator should be in Common LISP.

A solution to the seeming language problem was presented in the form of a Common LISP implementation of the Rete algorithm developed locally at Berkeley [Guerrieri87]. Although this implementation lacks some of the optimizing features present in OPS5, it allows for arbitrary data structures, test expressions in the conditions elements of rules, and user-defined functions. In addition, its relatively small size (1,400 lines of code compared to 3,200 lines for OPS5) has allowed easy code modifications for the needs of this project. One potential enhancement would be the inclusion of the object-oriented paradigm, based on Portable Common LOOPS (PCL), a partial implementation of the Common LISP Object System (CLOS) specification [Bobrow88].

Common LISP has proven to be an excellent, flexible language for the development of a prototype for the Expert Emulator. Advantageous features of the language include automatic memory management, superior handling of lists, patterns, and polymorphic data, a rich environment, and its interactive nature. Extensive use has been made of the `loop`

Iteration Macro [Symbolics85], a programmable iteration facility that looks like stylized English rather than LISP code and provides an array of constructs for iteration control, local variables, prologue and epilogue, and returned values.

In addition to the inference engine, the data-format translators (Section 4.4) and system-interface utilities have also been implemented in LISP. The specific language used has been Digital's implementation of Common LISP, VAXLISP [Digital86], Version U2.2. The program runs on VAX minicomputers (8600 series) and workstations (VAXStation II/GPX) under the ULTRIX Operating System (Digital's version of UNIX). The total size of the LISP version of the Expert Emulator is 2,500 lines of code.

6.2.3 Using CLIPS

The LISP version of the Expert Emulator demonstrated the feasibility of the new approach. Nevertheless, there was a performance penalty to be paid for the unoptimized code and the programming conveniences and interactive nature of LISP. The execution speed of anything but small problems is poor. Furthermore, the size of the executable programs is huge, since both the 3.5-megabyte VAXLISP executable and the 2.4-megabyte *suspended image* of the Expert Emulator are needed. A suspended image is a binary copy of the memory in use during an interactive LISP session. In this application, the compiled inference engine, rules, and other LISP functions are accessed through the suspended image of a compilation session performed previously. Running the LISP program requires large amounts of memory to accommodate the executables as well as the memory manager — LISP allocates memory automatically and reclaims unused cells when it runs out of memory (garbage collection).

As a result of the above performance problems, a second version of the Expert Emulator has been implemented using CLIPS [Giarratano89b], an expert-system shell developed by NASA. CLIPS, an acronym for C Language Integrated Production System, implements the Rete algorithm and supports forward chaining. Its capabilities are similar to those of OPS5, whereas syntactically it resembles a subset of ART. It is composed of 30,000 lines of C and has been ported to a wide variety of computers ranging from personal computers to Cray supercomputers.

Although the data structures and expressions supported by CLIPS are not as general as those of Common LISP, they are adequate for the particular application. Certain advanced capabilities of LISP, such as object-oriented programming through PCL, are not provided in CLIPS and must be substituted with additional repetitive code. However, CLIPS provides superb speed, good external-function interface, and several LISP-like predicates and functions.

Run-time performance results on examples presented in Chapter 3 have been compiled for the two versions of the Expert Emulator. Table 6.1 presents comparative CPU times for a VAX 8800 and Table 6.2 gives CPU times for a VAXStation II/GPX[1]. Results including program initialization times are shown in parentheses[2]. As illustrated, the speed-up obtained with CLIPS is over one order of magnitude and grows with the size of the problem. Table 6.3 summarizes the memory requirements of the two versions and shows gains of over one order of magnitude.

Example		CPU Time					
Name	Elements	LISP (sec)		CLIPS (sec)		Speed-up (×)	
BJT Oscillator	14	5.4	(6.4)	0.6	(3.8)	9.0	(1.7)
MOS Oscillator	21	26.0	(27.0)	1.5	(4.7)	17.3	(5.7)
Pull-Up	72	915.6	(916.6)	32.1	(35.3)	28.5	(25.9)

Table 6.1: Speed comparison of Common LISP and CLIPS on a VAX 8800

The data-format converters and system interface functions have been written using CLIPS rules. Although a procedural implementation would be faster, using rules gives acceptable speed. The size of the converters and system utilities is 1,800 lines of CLIPS code.

As a result of the performance improvement, the execution of more rules has become feasible between simulation runs (Section 4.6). For the LISP-based version only a single sequence of rule "firings" was allowed between simulation runs. In this way, the run-time overhead was kept within an acceptable range. The CLIPS version allows more

[1] LISP run times include time spent in garbage collection.
[2] CLIPS rules are not pre-compiled.

Example		CPU Time					
Name	Elements	LISP (sec)		CLIPS (sec)		Speed-up (×)	
BJT Oscillator	14	38.0	(45.5)	4.3	(24.1)	8.8	(1.9)
MOS Oscillator	21	191.9	(199.4)	10.1	(29.9)	19.0	(6.6)
Pull-Up	72	7629.8	(7637.3)	255.2	(275.0)	29.9	(27.8)

Table 6.2: Speed comparison of Common LISP and CLIPS on a VAXStation II/GPX

Object	Size		
	LISP (Mbytes)	CLIPS (Mbytes)	Reduction (×)
Executable Files	5.9	0.19	31
Virtual Memory	10–30	0.5–1	20

Table 6.3: Memory requirements of Common LISP and CLIPS

detailed examination of the available simulation input and output data, as well as multiple user queries, thus enhancing the corrective capabilities of the Expert Emulator.

6.3 Implementation of the Framework

Two different implementations of the NECTAR framework have been built: one for the alphanumeric interface and one for the windowed interface.

6.3.1 Using the UNIX Shell

The necessary operations to accomplish the framework tasks for an alphanumeric interface are the following:

- Interpretation of the user's commands
- Implementing the flow of control of the framework (Section 4.6)
- Invoking CAD tools and other programs
- Directing data between programs and databases (files)

- Controlling the execution of processes
- Database (file and directory) browsing
- Sending results and messages to the display

Almost all operations above involve in some way the underlying operating system (UNIX). All operations can be accomplished using UNIX's command interpreter (*shell*), the program that sits between the UNIX kernel and the user in a regular UNIX system.

The UNIX shell has several useful capabilities: it allows filename specification using shorthands and patterns; it can redirect the input and output of programs even without using files (through *pipes*); it monitors and controls processes; it allows the creation of new and complex commands through *aliases* and *shell files*. Furthermore, since the default output of the shell is the (alphanumeric) terminal, several facilities for the display of data and messages are available. Clearly, the UNIX shell is not a typical command interpreter. It is really a programming language with variables, loops, decision-making, etc. [Kernighan84].

Intended to serve as a prototype, the alphanumeric version of NECTAR was implemented using *shell scripts*, namely shell commands saved in a file. Such programs do not require compilation and allow rapid prototyping. The performance penalty that characterizes interpreted programs is not noticeable in NECTAR, since the program does not contain any computationally expensive loops. Consequently, there was no need to translate the prototype in a conventional language, such as C. As an added advantage, UNIX commands and programs can be accessed from inside the NECTAR shell.

Among the several versions of the UNIX shell, the one used in NECTAR has been `csh` [Joy83], a shell with C-like syntax. The size of this version of the framework is 1,100 lines of code.

6.3.2 Using the X Window System

The programming demands of a windowed interface are considerably higher than those of an alphanumeric interface. A multitude of additional operations are necessary for the windowed interface of NECTAR, including the following:

- Creating windows

- Positioning windows on the screen
- Redrawing and updating windows when changes occur
- Moving windows
- Resizing windows
- Combining simple windows to form complex structures
- Handling (selecting, editing, saving) text in text windows
- Scheduling events and managing the general behavior of all windows

Lengthy and complicated code is required to handle the added complexity. Characteristically, the "Hello World" program, the customary "minimal" program in the C language that can be written in less than 5 lines for an alphanumeric display, requires pages of code in windowed versions [Rosenthal87].

Windowing systems have been developed both to simplify programming and to set interface standards for application software. Whereas many windowing systems are specific to one type of hardware, some have been designed for generality and transparency. Among them, the *X Window System* (or simply X) [Scheifler86, Scheifler88] has achieved widespread popularity, particularly in the UNIX community. X provides high-performance, high-level, device-independent graphics. It is based on a *network protocol* and a client-server model: client programs running on the local workstation or on any machine on the network communicate with the *X server* program that runs on the workstation.

The windowed interface of NECTAR has been implemented for X, Protocol Version 10[3] [Gettys86]. The program (`xnectar`) has been written in the C Programming Language [Kernighan78] and runs on VAX computers under ULTRIX.

The complexity of windowing programs can be reduced significantly with the use of library packages, in the same way that *stdio* simplifies standard UNIX programming. For this application, the *X Toolkit Library* [Athena87] has been used. The X Toolkit extends the basic abstractions of X by providing a cohesive set of *widgets* and a component-interaction mechanism. Widgets are (sub)windows that provide certain user-interface abstractions (for example, a scroll-bar widget). The X Toolkit handles resizing and redrawing of widgets,

[3]Version 11 has since been released and become the new standard

text selection and editing in text windows, color defaults, and other window operations. Owing to the use of the X Toolkit, the size of the windowed interface has been kept to 2,200 lines of code.

Chapter 7

Conclusions

The research activity described in this dissertation has addressed the problem of improving existing analog CAD tools and the analog design process (Section 4.2.3) as a whole. In particular, the research objectives set in Chapter 2, Chapter 3, and Chapter 4 have been pursued. Along the way, issues drawn from different disciplines, including circuit simulation, circuit design, expert systems, computer frameworks, user interfaces, and programming languages, have been investigated, and solutions and methods have been proposed and implemented.

As its main thesis, this research has shown that *the use of* CAD *tools can be enhanced significantly without altering the tools themselves.* Improvements result from actions peripheral to the tools, including the following, as illustrated in Figure 7.1:

1. Appropriate modifications to the tool inputs, based on the application of expert rules
2. Automation of tool interaction, through their integration in a CAD framework
3. Simplification of tool invocation and result presentation, with the use of a uniform framework user interface that incorporates modern human-computer interaction principles.

NECTAR, the computer framework that implements the proposed improvement methods, offers solutions to several shortcomings commonly confronting analog designers. With the use of NECTAR during a typical design session, a number of new capabilities are attained:

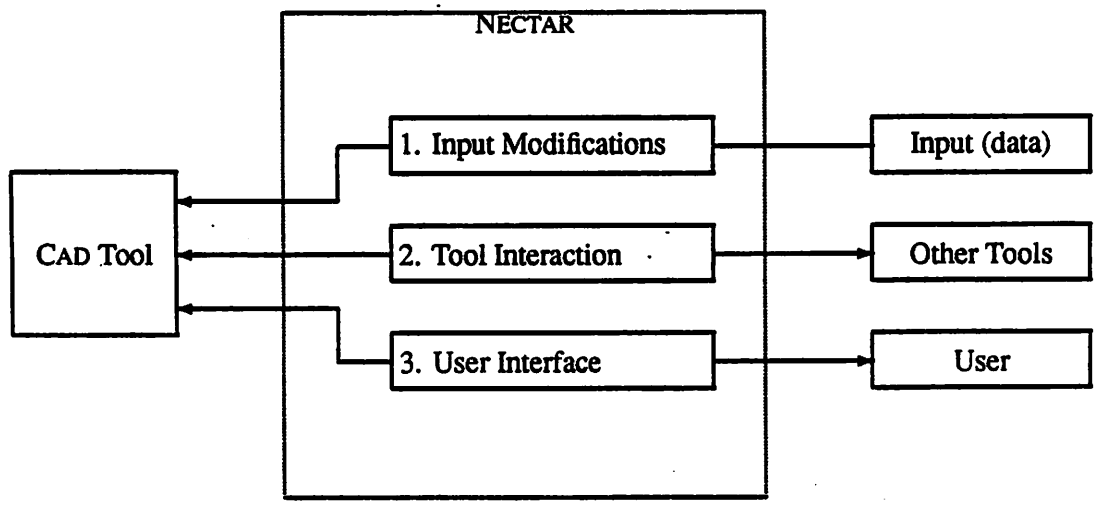


Figure 7.1: Tool-usage improvement methods implemented in NECTAR

Tool Access: By invoking the framework, the user gains *immediate* access to a host of otherwise "scattered" tools (simulators, post-processors, circuit editors, etc.) and utilities (for job monitoring and control, directory browsing, history, etc.) (Figure 4.3). Available capabilities are displayed either on the menu bar (Figure 5.5) or through the help facility (Figure 5.4).

Command Specification: Once inside NECTAR, the user can specify actions using simple and straightforward commands. Command-argument specification is flexible, with default values, command-line specification, and user queries being allowed. Side command requirements, such as initialization files, are filled automatically by NECTAR (Section 5.4).

Input (Design) Data: Design data generated with an editor is piped as input to another tool, e.g. a simulator, with no need to specify the name of the data file. In addition, input data is checked, upon request, for format and other errors (Section 3.5).

Job Monitoring: During a simulation run, partial results are displayed continuously (Section 5.4). The status of the job, in particular the elapsed CPU time, is also displayed (Figure 5.5). The job can be terminated with a simple command.

Output Data: Simulation results are stored in files automatically with no need for the user to specify file names (Section 5.4). Results from any simulator can be viewed using

any one of several post-processing and plotting programs (Section 4.3). The use of a common database by all tools eliminates redundant information flow and minimizes data-format translations (Section 4.4).

Simulation-Error Recovery: Hard-to-overcome simulation shortcomings, notably convergence problems, that otherwise require "trial-and-error" or expert-user consultation, can be solved on-line with the application of "rules-of-thumb" coded in rules (Section 3.4). New expert knowledge can be incorporated easily in NECTAR using a special rule editor (Section 5.7).

User Friendliness: A user of NECTAR can benefit from a user interface designed with the user in mind. System and tool details are hidden and emphasis has been put on tasks, not tools (Section 5.3.2). NECTAR provides a uniform interface for several (individually polymorphic) tools (Section 5.4). Complicated command-control sequences, frequently requested by users, are incorporated in the framework flow of control (Section 4.6).

Hardware: NECTAR allows the use of different types of computer hardware. Jobs can run on remote machines to optimize CPU usage (Section 4.5). Two different interfaces are available that correspond to the two most widely used types of terminals. The two interfaces are compatible in the tasks that can be accomplished and, therefore, work started on one type of terminal can be continued using a different type (Section 5.3.3).

Although only the third improvement action mentioned at the beginning of this chapter refers to what is considered user interface in the strict sense, the other two actions also improve the interaction with existing tools. Hence, NECTAR can be considered a user-interface shell, an *auxiliary* tool that can provide significant help to users of analog-circuit analysis and design programs.

Appendix A

A Short Chronological Review of SPICE

By the end of the 60's a major research activity was present at the University of California at Berkeley (UCB) and involved the investigation and development of several circuit simulation programs [Pederson84]. One project, involving ten students under Rohrer, tackled most aspects of computer circuit analysis and resulted in the Program CANCER [Nagel71]. The first version of CANCER came out during the Academic Year 1969–70 and included nonlinear DC and AC analyses. By the Fall of 1970 Nagel had added the capability for nonlinear transient analysis. Versions 3, 4, and 5 of CANCER were used in instructional courses during 1970–72 at UCB. It should be noted that CANCER was never released in the public domain.

In 1971 the SPICE project was initiated by Nagel and Pederson. The first version was released in the public domain in May 1972. Several versions of SPICE1 were released in the next three years. In July of 1975 SPICE2A.1, the first version of SPICE2 [Nagel75, Cohen76], was released. For all versions up to 2E UCB used the CDC6000 computer, a 64-bit machine. While others had converted to 32-bit machines earlier, UCB moved to a 32-bit machine with Version 2F, which was developed for a VAX running UNIX by Dowell, Newton, and Vladimirescu and was released in March 1980. From July 1980 to August 1983 the G versions were released [Vladimirescu81]. 2G.0 included a source-stepping algorithm, and 2G.6 was the last release of SPICE2, in 1983. Versions 2G.7 and 2G.8 were completed in February of 1984 but were not released because of the upcoming SPICE3. SPICE2G.6 is still used widely in the academia and industry.

The SPICE3 project was spurred by the need for a better structured, more modular software package. Written in FORTRAN and already over 10 years old by the early 80's, SPICE2 impeded easy program modifications and enhancements. As an example, the addition of a new device model into SPICE2 constituted a much more complicated problem than one might think or desire. The first version of SPICE3 was written in RATFOR by Quarles in December of 1983 [Quarles83]. After that, the C Programming Language was chosen for the project. Quarles wrote the core routines and Christopher implemented the front-end and post-processor program, called Nutmeg [Christopher87]. The three main releases of SPICE3 have been: 3a1 in March 1985, 3b1 in April 1987, and 3c1 in April 1989. Compared with SPICE2, SPICE3 has cleaner data structures, is more modular, and has a friendlier user interface. New models, as well as new analyses, can be (and have been) added with considerably less effort than for SPICE2. Nevertheless, SPICE3 has not yet replaced SPICE2, which is still widely used. A more complete review of the evolution of SPICE, including the many commercial derivatives, can be found in [Vladimirescu90].

Appendix B

Error Messages in SPICE

This appendix lists the error messages and warnings generated by SPICE2 in response to run-time problems. These messages are often not sufficiently informative and sometimes refer to entities internal to the program. Other messages include corrective suggestions. Fatal errors are distinguished by the "*ERROR*" prefix and are listed first. Warnings may not cause the termination of the simulation job; they are prefixed with "WARNING." In the following, the character '*X*' denotes variables whose values are determined at run time.

Error Messages

ERROR: CPU TIME LIMIT EXCEEDED ... ANALYSIS STOPPED

ERROR: PARAMETER CHANGE FAILED *X* IS NOT IN THE ORIGINAL CIRCUIT

ERROR: .END CARD MISSING

ERROR: ILLEGAL NUMBER – SCAN STOPPED AT COLUMN *X*

ERROR: MAXIMUM ENTRY IN THIS COLUMN AT STEP *X* (*X*) IS LESS THAN PIVTOL

ERROR: NO CONVERGENCE IN DC ANALYSIS. LAST NODE VOLTAGES:

ERROR: NO CONVERGENCE IN DC TRANSFER CURVES AT *X* = *X*. LAST NODE VOLTAGES:

ERROR: TEMPERATURE SWEEP SHOULD BE THE SECOND SWEEP SOURCE, CHANGE THE ORDER AND RE-EXECUTE

ERROR: INTERNAL TIMESTEP TOO SMALL IN TRANSIENT ANALYSIS

ERROR: TRANSIENT ANALYSIS ITERATIONS EXCEED LIMIT OF *X*. THIS LIMIT MAY BE OVERRIDDEN USING THE ITL5 PARAMETER ON THE .OPTION CARD

ERROR: CPU TIME LIMIT EXCEEDED IN TRANSIENT ANALYSIS AT TIME = *X*

ERROR: *X* HAS BEEN REFERENCED BUT NOT DEFINED

ERROR: CIRCUIT HAS NO NODES

ERROR: ELEMENT *X* PIECEWISE LINEAR SOURCE TABLE NOT INCREASING IN TIME

ERROR: MEMORY REQUIREMENT EXCEEDS MACHINE CAPACITY MEMORY NEEDS EXCEED *X* (O6B)

ABORT: INTERNAL MEMORY MANAGER ERROR AT ENTRY *X*

ERROR: ABOVE LINE ATTEMPTS TO REDEFINE *X*

ERROR: UNABLE TO FIND *X*

ERROR: SYSTEM ERROR, ADDRESS *X* IS NOT ON 4-BYTE BOUNDARY

ERROR: NSUB <= NI IN MOSFET MODEL *X*

ERROR: EFFECTIVE CHANNEL LENGTH OF *X* LESS THAN ZERO. CHECK VALUE OF LD FOR MODEL *X*

ERROR: UNKNOWN DATA CARD: *X*

ERROR: UNRECOGNIZABLE DATA CARD

ERROR: Z0 MUST BE SPECIFIED

ERROR: EITHER TD OR F MUST BE SPECIFIED

ERROR: ELEMENT TYPE NOT YET IMPLEMENTED

ERROR: NEGATIVE NODE NUMBER FOUND

ERROR: NODE NUMBERS ARE MISSING

ERROR: VALUE IS MISSING OR IS NONPOSITIVE

ERROR: MUTUAL INDUCTANCE REFERENCES ARE MISSING

ERROR: MODEL NAME IS MISSING

ERROR: UNKNOWN SOURCE FUNCTION: *X*

ERROR: UNKNOWN PARAMETER: *X*

ERROR: VOLTAGE SOURCE NOT FOUND ON ABOVE LINE

ERROR: VALUE IS ZERO

ERROR: EXTRA NUMERICAL DATA ON MOSFET CARD

ERROR: MODEL TYPE IS MISSING

ERROR: UNKNOWN MODEL TYPE: *X*

ERROR: UNKNOWN MODEL PARAMETER: *X*

ERROR: SUBCIRCUIT DEFINITION DUPLICATES NODE *X*

ERROR: NONPOSITIVE NODE NUMBER FOUND IN SUBCIRCUIT DEFINITION

ERROR: SUBCIRCUIT NAME MISSING

ERROR: SUBCIRCUIT NODES MISSING

ERROR: UNKNOWN SUBCIRCUIT NAME: *X*

ERROR: SUBCIRCUIT NAME MISSING

ERROR: .ENDS CARD MISSING

ABORT: SPICE INTERNAL ERROR IN REORDR

ABORT: INTERNAL SPICE ERROR: SORUPD: *X*

ERROR: *X* HAS DIFFERENT NUMBER OF NODES THAN *X*

ERROR: SUBCIRCUIT *X* IS DEFINED RECURSIVELY

ERROR: LESS THAN 2 CONNECTIONS AT NODE *X*

ERROR: NO DC PATH TO GROUND FROM NODE *X*

ERROR: INDUCTOR/VOLTAGE SOURCE LOOP FOUND, CONTAINING *X*

***** JOB ABORTED

Warnings

WARNING: UNDERFLOW *X* TIME(S) IN AC ANALYSIS AT FREQ = *X* HZ

WARNING: UNDERFLOW *X* TIME(S) IN DISTORTION ANALYSIS AT FREQ = *X* HZ

WARNING: MORE THAN *X* POINTS FOR *X* ANALYSIS, ANALYSIS OMITTED. THIS LIMIT MAY BE OVERRIDDEN USING THE LIMPTS PARAMETER ON THE .OPTION CARD

WARNING: NO *X* OUTPUTS SPECIFIED ... ANALYSIS OMITTED

WARNING: FOURIER ANALYSIS FUNDAMENTAL FREQUENCY IS INCOMPATIBLE WITH TRANSIENT ANALYSIS PRINT INTERVAL ... FOURIER ANALYSIS OMITTED

WARNING: ATTEMPT TO REFERENCE UNDEFINED NODE *X* – NODE RESET TO 0

WARNING: UNDERFLOW OCCURRED *X* TIME(S)

WARNING: MINIMUM BASE RESISTANCE (RBM) IS LESS THAN TOTAL (RB) FOR MODEL *X*. RBM SET EQUAL TO RB

WARNING: THE VALUE OF LAMBDA FOR MOSFET MODEL *X* IS UNUSUALLY LARGE AND MIGHT CAUSE NONCONVERGENCE

WARNING: IN DIODE MODEL *X* IBV INCREASED TO *X*, TO RESOLVE INCOMPATIBILITY WITH SPECIFIED IS

WARNING: UNABLE TO MATCH FORWARD AND REVERSE DIODE REGIONS BV = *X* AND IBV = *X*

WARNING: TOO FEW POINTS FOR PLOTTING

WARNING: INPUT LINE-WIDTH SET TO 72 COLUMNS BECAUSE POSSIBLE SEQUENCING APPEARS IN COLS 73-80

WARNING: ABOVE LINE NOT ALLOWED WITHIN SUBCIRCUIT – IGNORED

WARNING: COEFFICIENT OF COUPLING RESET TO 1.0D0

WARNING: NO SUBCIRCUIT DEFINITION KNOWN – LINE IGNORED

WARNING: MISSING PARAMETER(S) ... ANALYSIS OMITTED

WARNING: UNKNOWN FREQUENCY FUNCTION: *X* ... ANALYSIS OMITTED

WARNING: FREQUENCY PARAMETERS INCORRECT ... ANALYSIS OMITTED

WARNING: START FREQ > STOP FREQ ... ANALYSIS OMITTED
WARNING: TIME PARAMETERS INCORRECT ... ANALYSIS OMITTED
WARNING: START TIME > STOP TIME ... ANALYSIS OMITTED
WARNING: ILLEGAL OUTPUT VARIABLE ... ANALYSIS OMITTED
WARNING: VOLTAGE OUTPUT UNRECOGNIZABLE ... ANALYSIS OMITTED
WARNING: INVALID INPUT SOURCE ... ANALYSIS OMITTED
WARNING: DISTORTION LOAD RESISTOR MISSING ... ANALYSIS OMITTED
WARNING: DISTORTION PARAMETERS INCORRECT ... ANALYSIS OMITTED
WARNING: FOURIER PARAMETERS INCORRECT ... ANALYSIS OMITTED
WARNING: OUTPUT VARIABLE UNRECOGNIZABLE ... ANALYSIS OMMITTED
WARNING: NUMDGT MAY NOT EXCEED *X* ; MAXIMUM VALUE ASSUMED
WARNING: UNKNOWN OPTION: *X* ... IGNORED
WARNING: ILLEGAL VALUE SPECIFIED FOR OPTION: *X* ... IGNORED
WARNING: UNKNOWN ANALYSIS MODE: *X* ... LINE IGNORED
WARNING: UNRECOGNIZABLE OUTPUT VARIABLE ON ABOVE LINE
WARNING: OUT-OF-PLACE NON-NUMERIC FIELD *X* SKIPPED
WARNING: INITIAL VALUE MISSING FOR NODE *X*
WARNING: ATTEMPT TO SPECIFY INITIAL CONDITION FOR GROUND INGNORED
WARNING: OUT-OF-PLACE NON-NUMERIC FIELD *X* SKIPPED
WARNING: INITIAL VALUE MISSING FOR NODE *X*
WARNING: ATTEMPT TO SPECIFY INITIAL CONDITION FOR GROUND IGNORED
WARNING: FURTHER ANALYSIS STOPPED DUE TO CPU TIME LIMIT

Appendix C

Program Source Listing

The source listing of NECTAR is available at the following address:

Software Distribution Office
Industrial Liaison Program
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

The following should be noted:

- The LISP version of the Expert Emulator requires the executable of VAXLISP, version U2.2, which can be obtained from Digital Equipment Corporation.
- The CLIPS version of the Expert Emulator requires the executable of CLIPS, version 4.3, which can be obtained from COSMIC, The University of Georgia, Athens, GA 30602.
- The X-Windows version of NECTAR runs under version 10 of the X protocol.

Bibliography

[Acuna90] E. Acuna, J. Dervenis, A. Pagones, F. Yang, and R. Saleh, "Simulation techniques for mixed analog/digital circuits," *IEEE Journal of Solid-State Circuits*, vol. SC-25, no. 2, pp. 353–363, April 1990.

[Allen86] P. Allen, "A tutorial — computer aided design of analog integrated circuits," *Proc. of the IEEE 1986 Custom Integrated Circuits Conference*, pp. 608–616, Rochester, New York, May 1986.

[Apple85] Apple Computer, Inc., *Inside Macintosh,* Cupertino, California, 1985.

[Athena87] MIT Project Athena, "The X toolkit intrinsics," *MIT Project Athena Document,* November 1987.

[Ashar89] P. Ashar, "Implementation of algorithms for the periodic-steady-state analysis of nonlinear circuits," *Memo No. UCB/ERL-M89/31*, University of California, Berkeley, March 1989.

[Barr82] A. Barr and E. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, Heuristech Press, 1982.

[Bergquist86] S. Bergquist and R. Sparkes, "QCritic: a rule-based analyzer for bipolar analog circuit designs," *Proc. of the IEEE 1986 Custom Integrated Circuits Conference*, pp. 617–620, Rochester, New York, May 1986.

[Bobrow88] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and A. Moon, "Common LISP object system specification," *Technical Report X3J13 Document 88-002R,* American National Standards Institute, June 1988.

[Bradley87] J. Bradley, "Xsplot user's manual," *UNIX Manual Page*, University of Pennsylvania, 1987.

[Brodersen84] R. Brodersen, "The future role of analog circuits in custom circuits," *Proc. of the IEEE 1984 Custom Integrated Circuits Conference*, pp. 594–595, Rochester, New York, May 1984.

[Brown83] H. Brown, C. Tong, and G. Foyster, "Palladio: an exploratory environment for circuit design," *IEEE Computer*, vol. 16, no. 12, pp. 41–56, December 1983.

[Brownston85] L. Brownston, R. Farrell, E. Kant, N. Martin, *Programming expert systems in OPS5*, Addison-Wesley, 1985.

[Bushnell87] M. Bushnell, "ULYSSES—an expert-system based VLSI design environment," *Ph.D. Thesis*, Carnegie-Mellon University, ECE Dept., May 1987.

[Cande86] CANDE Committee, "CANDE predictions for early 1990's," April 1986.

[Carley88] R. Carley and R. Rutenbar, "How to automate analog IC designs," *IEEE Spectrum*, vol. 25, no. 8, pp. 26–30, August 1988.

[Christopher87] W. Christopher, "Nutmeg," *Berkeley CAD Tools User's Manual*, University of California, Berkeley, EECS Dept./ERL, April 1987.

[Christopher87b] W. Christopher, "Sconvert," *Berkeley CAD Tools User's Manual*, University of California, Berkeley, EECS Dept./ERL, April 1987.

[Cobourn87] T. Cobourn, "An evaluation of Cleopatra, a natural language interface for CAD," *Research Report CMUCAD-87-1*, Department of Electrical and Computer Engineering, Carnegie-Mellon University, January 1987.

[Cohen76] E. Cohen, "Program reference for SPICE2," *Memo No. ERL-M592*, University of California, Berkeley, EECS Dept./ERL, June 1976.

[Colon89] R. Colon, *Private Communication*, September 1989.

[Daniell89] J. Daniell, "An object oriented approach to CAD tool control," *Research Report CMUCAD-89-37*, Carnegie-Mellon University, April 1989.

[Degrauwe87] M. Degrauwe, O. Nys, E. Dijkstra, J. Rijmenants, S. Bitz, B. Goffart, E. Vittoz, S. Cserveny, C. Meixenberger, G. van der Stappen, and H. Oguey, "IDAC: an interactive design tool for analog CMOS circuits," *IEEE Journal of Solid-State Circuits*, vol. SC-22, no. 6, pp. 1106–1116, Decemeber 1987.

[DeMan85] H. De Man, I. Bolsens, E. Vanden Meersch, J. Van Cleynenbreugel, "DIALOG: an expert debugging system for MOS VLSI design," *IEEE Transactions on Computer-Aided Design*, vol. 4, no. 3, July 1985.

[Digital86] Digital Equipment Co., *VAX LISP/ULTRIX user's guide*, Maynard, Massachusetts, May 1986.

[ElTurky89] F. El-Turky and E. Perry, "BLADES: an artificial intelligence approach to analog circuit design," *IEEE Trans. on Computer-Aided Design*, vol. 8, no. 6, June 1989.

[Engl82] W. L. Engl, R. Laur, and H. K. Dirks, "MEDUSA—a simulator for modular circuits," *IEEE Transactions on Computer-Aided Design*, vol. CAD-1, no. 2, pp. 85–93, April 1982.

[Foderaro83] J. Foderaro, K. Sklower, and K. Layer, "The Franz LISP manual," University of California, Berkeley, June 1983.

[Forgy81] C. Forgy, "OPS5 user's manual," *Technical Report CMU-CS-81-135*, Department of Computer Science, Carnegie-Mellon University, July 1981.

[Forgy82] C. Forgy, "Rete: a fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, vol. 19(1), pp. 17–37, September 1982.

[Forgy84] C. Forgy, "The OPS83 report," *Technical Report CMU-CS-84-133*, Department of Computer Science, Carnegie-Mellon University, May 1984.

[Forsythe77] G. Forsythe, M. Malcolm, nad C. Moler, *Computer methods for mathematical computations*, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[Friedenson82] R. Friedenson, J. Breiland, and T. Thompson, "Designer's Workbench: delivery of CAD tools," *Proc. of the 19th Design Automation Conference*, 1982.

[Gettys86] J. Gettys, R. Newman, and T. Della Fera, "Xlib — C language X interface, protocol version 10," *MIT Project Athena Document*, November 1986.

[Giarratano89] J. Giarratano and G. Riley, *Expert systems: principles and programming*, PWS-Kent Publishing Co., Boston, 1989.

[Giarratano89b] J. Giarratano, "CLIPS user's guide, version 4.3 of CLIPS," *Program Documentation*, NASA L. B. Johnson Space Center, AI Section, May 1989.

[Granacki85] J. Granacki, D. Knapp, and A. Parker, "The ADAM advanced design automation system: overview, planner, and natural language interface," *Proc. of the 22nd Design Automation Conference*, June 1985.

[Gray84] P. Gray and R. Meyer, *Analysis and design of analog integrated circuits*, Second Edition, Wiley, 1984.

[Guerrieri87] R. Guerrieri, *Private Communication*, September 1987.

[Hachtel71] G. Hachtel, R. Brayton, and F. Gustavson, "The sparse tableau approach to network analysis and design," *IEEE Transactions on Circuit Theory*, vol. CT-18, pp. 101–113, January 1971.

[Hachtel81] G. Hachtel and A. Sangiovanni-Vincetelli, "A survey of third-generation simulation techniques," *IEEE Proceedings*, vol. 69, pp. 1264–1280, October 1981.

[Hailey87] S. Hailey, *Private Communication*, 1987.

[Harjani87] R. Harjani, R. Rutenbar, and R. Carley, "A prototype framework for knowledge-based analog circuit synthesis," *Technical Report CMUCAD-87-26*, Department of Electrical and Computer Engineering, Carnegie-Mellon University, June 1987.

[Harrison86] D. Harrison, P. Moore, R. Spickelmier, and R. Newton, "Data management and graphics editing in the Berkeley design environment," *Proc. of the International Conference on Computer-Aided Design*, Santa Clara, California, November 1986.

[Harrison88] D. Harrison, "Xgraph user's manual," *UNIX Manual Page*, University of California, Berkeley, EECS Dept./ERL, April 1988.

[Harrison89] D. Harrison, "VEM: interactive graphics for Oct," *Master's Thesis*, University of California, Berkeley, EECS Dept., 1989.

[Heckel84] P. Heckel, *The elements of friendly software design*, Warner Books, 1984.

[Ho75] C. Ho, A. Ruehli, and P. Brennan, "The modified nodal approach to network analysis," *IEEE Transactions on Circuits and Systems*, vol. CAS-22, no. 6, pp. 504–509, June 1975.

[Hodges88] D. Hodges and H. Jackson, *Analysis and design of digital integrated circuits*, Second Edition, McGraw-Hill, 1988.

[Inference89] Inference Corp., "ART Programming Tutorial," Los Angeles, California, 1989.

[Johnson89] J. Johnson, T. Roberts, W. Verplank, D. Smith, C. Irby, M. Beard, and K. Mackey, "The Xerox Star: a retrospective," *IEEE Computer*, vol. 22, no. 9, pp. 11–28, September 1989.

[Joy83] W. Joy, "Csh," *UNIX programmer's manual*, University of California, Berkeley, CS Div., EECS Dept., July 1983.

[Kelly84] V. Kelly, "The CRITTER system: automated critiquing of digital circuit designs," *Proc. of the 21st Design Automation Conference*, Albuquerque, New Mexico, 1984.

[Kernighan78] B. Kernighan and D. Ritchie, *The C programming language*, Prentice-Hall, 1978.

[Kernighan84] B. Kernighan and B. Pike, *The UNIX programming environment*, Prentice-Hall, 1984.

[Koh90] H. Koh, C. Séquin, and P. Gray, "OPASYN: a compiler for CMOS operational amplifiers," *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 2, pp. 113–125, February 1990.

[Kowalski85] T. Kowalski, *An artificial intelligence approach to VLSI design*, Kluwer Academic Publishers, Boston, 1985.

[Kundert86] K. Kundert and A. Sangiovanni-Vincentelli, "Simulation of nonlinear circuits in the frequency domain," *IEEE Transactions on Computer-Aided Design*, vol. CAD-5, no. 4, pp. 521–535, October 1986.

[Kuo90] A. Kuo, "SPLINT," *Program Documentation*, University of California, Berkeley, May 1990.

[Laidig90] T. Laidig, "Octspice," *Berkeley CAD Tools User's Manual*, University of California, Berkeley, EECS Dept./ERL, February 1990.

[Lob84] C. Lob, "RUBICC: a rule-based expert system for VLSI integrated circuit critique," *Memo No. ERL-M84/80*, University of California, Berkeley, September 1984.

[Markov54] A. Markov, *Theory of algorithms (Teoriya algorifmov)*, Academy of Sciences of the USSR, Moskow, 1954, (translated from Russian by J. Schorr-Kon and IPST staff, Israel Program for Scientific Translations, Jerusalem, 1971).

[Mayaram87] K. Mayaram and D. O. Pederson, "Analysis of MOS transformer-coupled oscillators," *IEEE Journal of Solid-State Circuits*, vol. SC-22, no. 6, December 1987.

[Mayaram88] K. Mayaram, "CODECS: a mixed-level circuit and device simulator," *Memo No. UCB/ERL-M88/71*, University of California, Berkeley, November 1988.

[McCalla88] W. McCalla, *Fundamentals of computer-aided circuit simulation*, Kluwer Academic Publishers, Boston, 1988.

[McDermott78] J. McDermott, A. Newell, and J. Moore, "The efficiency of certain production system implementations," in *Pattern-directed inference systems*, ed. by D. Waterman and F. Hayes-Roth, pp. 155–176, Academic Press, New York, 1978.

[Mead80] C. Mead and L. Conway, *Introduction to VLSI systems*, Addison-Wesley, 1980.

[Meyer71] J. Meyer, "MOS models and circuit simulation," *RCA Review*, vol. 32, pp. 42–63, March 1971.

[Moran81] T. Moran, "The Command Language Grammar: a representation for the user interface of interactive computer systems," *International Journal of Man-Machine Studies*, vol. 15, no. 1, pp. 3–50, July 1981.

[Nagel71] L. Nagel and R. Rohrer, "Computer analysis of nonlinear circuits excluding radiation (CANCER)," *IEEE Journal of Solid-State Circuits*, vol. SC-6, no. 4, pp. 166-182, August 1971.

[Nagel75] L. Nagel, "SPICE2, a computer program to simulate semiconductor circuits," *Memo No. ERL-M520*, University of California, Berkeley, May 1975.

[Newton84] A.R. Newton and A. Sangiovanni-Vincentelli, "Relaxation-based electrical simulation," *IEEE Transactions on Computer-Aided Design*, vol. CAD-3, no. 4, pp. 308–331, October 1984.

[Nickerson90] R. Nickerson and R. Pew, "Toward more compatible human-computer interfaces," *IEEE Spectrum*, vol. 27, no. 7, pp. 40–43, July 1990.

[Nye88] W. Nye, D. Riley, A. Sangiovanni-Vincentelli, and A. Tits, "DELIGHT.SPICE: an optimization-based system for the design of integrated circuits," *IEEE Transactions on Computer-Aided Design*, vol. 7, no. 4, pp. 501–519, April 1988.

[Pederson84] D. O. Pederson, "A historical review of circuit simulation," *IEEE Transactions on Circuits and Systems*, vol. CAS-31, no. 1, pp. 103–111, January 1984.

[Pederson90] D. O. Pederson, *Private communication*, April 1990.

[Pederson90b] D. O. Pederson and K. Mayaram, *Analog integrated circuits for communications: principles, simulation and design*, Kluwer Academic Publishers, 1990.

[Post43] E. Post, "Formal reductions of the general combinatorial decision problem," *American Journal of Mathematics*, vol. 65, pp. 197–215, 1943.

[Price82] C. H. Price, "Two-dimensional numerical simulation of semiconductor devices," *Ph.D. Dissertation*, Stanford University, Stanford, 1982.

[Quarles83] T. Quarles, "The SPICE3 circuit simulator," *Memo No. ERL-M592*, University of California, Berkeley, December 1983.

[Quarles89] T. Quarles, "Analysis of performance and convergence issues for circuit simulation," *Memo No. UCB/ERL M89/42*, University of California, Berkeley, EECS Dept./ERL, April 1989.

[Quarles89b] T. Quarles, "SPICE3 version 3C1 user's guide," *Memo No. UCB/ERL M89/46*, University of California, Berkeley, April 1989.

[Ralston78] A. Ralston and P. Rabinowitz, *A first course in numerical analysis*, Second Edition, McGraw-Hill, New York, 1978.

[Rosenthal87] D. Rosenthal, "A simple X.11 client program or how hard can it really be to write "Hello World"? ," Sun Microsystems, Inc., San Jose, California, 1987.

[Saleh87] R. Saleh, "Nonlinear relaxation algorithms for circuit simulation," *Memo No. UCB/ERL M87/21*, Electronics Research Laboratory, University of California, Berkeley, April 1987.

[Samad86] T. Samad, "Towards a natural language interface for computer aided design," *Ph.D. Thesis*, Carnegie-Mellon University, ECE Dept., January 1986.

[Sangiovanni81] A. Sangiovanni-Vincentelli, "Circuit simulation," in *Computer design aids for VLSI circuits*, ed. by P. Antognetti, D. O. Pederson, and H. De Man, pp, 19–113, Sijthoff and Noordboff, Groningen, The Netherlands, 1981.

[Scheifler86] R. Scheifler and J. Gettys, "The X window system," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79–109, April 1986.

[Scheifler88] R. Scheifler, J. Gettys, and R. Newman, *X window system: C library and protocol reference* Digital Press, Bedford, Massachusetts, 1988.

[Shneiderman87] B. Shneiderman, *Designing the user interface*, Addison-Wesley, 1987.

[Siewiorek84] D. Siewiorek, D. Giuse, W. Birmingham, M. Hirsch, V. Rao, and G. York, "DEMETER project: phase 1 (1984)," *Research Report CMUCAD-84-35*, Department of Electrical and Computer Engineering, Carnegie-Mellon University, July 1984.

[Shyu88] J. Shyu, "Performance optimization of integrated circuits," *Memo No. UCB/ERL-M88/74*, University of California, Berkeley, November 1988.

[Smith82] S. Smith, "User-interface design for computer-based information systems," *Report ESD-TR-82-132*, The MITRE Corporation, Bedford, Massachusetts, April 1982.

[Spickelmier89] R. Spickelmier, "Using knowledge-based systems for circuit critiquing," *Ph.D. Dissertation*, University of California, Berkeley, EECS Dept., October 1989.

[Steele84] G. Steele, *Common LISP: the language*, Digital Press, 1984.

[Stefik82] M. Stefik, J. Aikins, R. Balzer, J. Benoit, L. Birnbaum, F. Hayes-Roth, and E. Sacerdoti, "The organization of expert systems, a tutorial," *Artificial Intelligence*, vol. 18, pp. 135–173, 1982.

[Stefik86] M. Stefik and D. Bobrow, "Object-oriented programming: themes and variations," *The AI Magazine*, vol. 6, no. 4, pp. 40–62, Winter 1986.

[Symbolics85] The Documentation Group of Symbolics, Inc., "Reference guide to Symbolics-LISP," *Symbolics Document # 996025*, Cambridge, Massachusettes, June 1985.

[Tsividis87] Y. Tsividis, *Operation and modeling of the MOS transistor*, McGraw-Hill, 1987.

[Udell90] J. Udell, "Three's the one," *Byte*, vol. 15, no. 6, pp. 122–128, June 1990.

[Vladimirescu80] A. Vladimirescu and S. Liu, "The simulation of MOS integrated circuits using SPICE2," *Memo No. UCB/ERL-M80/7*, University of California, Berkeley, February 1980.

[Vladimirescu81] A. Vladimirescu, K. Zhang, A. R. Newton, D. O. Pederson, A. Sangiovanni-Vincentelli, *SPICE version 2G users's guide*, University of California, Berkeley, EECS Dept./ERL, 1981.

[Vladimirescu90] A. Vladimirescu, "SPICE — the third decade," *Proc. 1990 Bipolar Circuits and Technology Meeting*, Minneapolis, Minn., September 1990 (to appear).

[Wærn89] Y. Wærn, *Cognitive aspects of computer supported tasks*, Wiley, 1989.

[Walker86] A. Walker, "Knowledge systems: Principles and practice," *IBM Journal of Research and Development*, vol. 30, no. 1, January 1986.

[Walters88] J. Walters, "An explanation of frame-based reasoning," *Seminar Talk*, Santa Clara, California, February 1988.

[Weeks73] W. Weeks, A. Jimenez, G. Mahoney, D. Mehta, H. Qassemzadeh, and T. Scott, "Algorithms for ASTAP—a network-analysis program," *IEEE Transactions on Circuit Theory*, vol. CT-20, no. 6, pp. 628–634, November 1973.

[White86] J. White and A. Sangiovanni-Vincentelli, *Relaxation techniques for the simulation of VLSI circuits*, Kluwer Academic Publishers, 1986.

[Wilensky84] R. Wilensky, *LISPcraft*, W. W. Norton & Co., New York, 1984.

[Williams83] G. Williams, "The Lisa computer system," *Byte*, vol. 8, no. 2, pp. 33–50, February 1983.