Exploring Semiconductor Device Parameter Space using Rapid Analytical Modeling

by

Brian Lee


Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1998

©1998 Massachusetts Institute of Technology
All rights reserved

Signature of Author: ____________________________

Department of Electrical Engineering and Computer Science
January 30, 1998

Certified by: ____________________________

Duane S. Boning
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by: ____________________________

Michael B. McIlrath
Research Scientist, Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: ____________________________

Arthur E. Smith
Chair
Department Committee on Graduate Students

MAR 27 1998
Exploring Semiconductor Device Parameter Space using Rapid Analytical Modeling

by

Brian Lee

Submitted to the Department of Electrical Engineering and Computer Science
on January 30, 1998 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in
Electrical Engineering and Computer Science

Abstract

The creation and implementation of a semiconductor device parameter space exploration tool is described, which can assist circuit and device designers. Such a tool allows the designer to analyze trade-offs between parameter variations (after defining electrical constraints at the circuit level of abstraction) as well as receive information about feasible device structures. In addition, it is also be possible to compare device parameter variations within a technology family, as well as across different technology families.

The technique of exploring parameter space utilizes a method of rapid analytical modeling to allow for faster, but less accurate, evaluations than one might get through established methods of numerical simulation tools such as MEDICI or PISCES, offering an alternative to such simulators for circuit designers who wish to have rough estimates of parameter variation information, or device structure feasibility. The tool can be a valuable addition to any circuit or device designer's CAD environment.

Thesis Supervisor: Duane S. Boning
Title: Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Michael B. McIlrath
Title: Research Scientist
Acknowledgments

To my thesis supervisors, Duane Boning and Michael McIlrath; for providing me with tons of useful suggestions and information for my research work.

To my academic and research advisor Professor Troxel; who got me started here at MIT with his kind offer of a research assistantship, and has provided constant support and advice along the way.

To my family; for always being there and for letting me find my way in the world.

To my officemates past and present: William Moyne, Thomas Lohman, Yonald Chery, Matthew Verminski, Manuel Perez, Erik Pedersen, Jared Cottrell and Francis Doughty; for providing a stimulating and enjoyable place to work.

This work has been made possible by the support from the Defense Advanced Research Projects Agency (DARPA) contract # DABT 63-95-C-0088.
# Table of Contents

## Chapter 1 Introduction

1. Device Parameter Space Exploration .................................................. 10
   1.1 Semiconductor Device Parameter Classifications .......................... 11
2. Definition of Problem ........................................................................ 11
3. Motivation .......................................................................................... 12
4. Device-Parameter Space Exploration Tool (DSET) .............................. 16
5. Applications ...................................................................................... 16
   5.1 Feasible Device Parameter Generation ........................................... 16
   5.2 Exploration of Future Technologies ................................................. 17
   5.3 Sensitivity Analysis ........................................................................ 17
   5.4 Technology Comparisons ............................................................ 17
6. Thesis Overview .................................................................................. 18

## Chapter 2 System Design

2. System Components ........................................................................... 20
   2.1 Main .......................................................................................... 21
   2.2 Model Objects ............................................................................ 21
   2.3 Model Library Manager ............................................................... 21
   2.4 User Input Module ....................................................................... 22
   2.5 Exploration Engine ...................................................................... 22
   2.6 Optimizer .................................................................................... 25
   2.7 Output Display ............................................................................ 26
2. System Operation ................................................................................ 27

## Chapter 3 Device Models

3. Device Models .................................................................................... 31
   3.1 Bulk CMOS ................................................................................ 31
   3.2 Silicon-on-Insulator with Active Substrate (SOIAS) ....................... 34
   3.3 Silicon-Germanium (Si-Ge) .......................................................... 36
3.2 Calibration and Fitting Issues .................................................................37
3.3 Model Objects - Protocol ........................................................................38
  3.3.1 Functional Characteristics ..................................................................38
  3.3.2 Device Model-related Characteristics .................................................39
3.4 Optimization Information Object Protocol ..........................................40
  3.4.1 Properties List ..................................................................................41
  3.4.2 Function Objects List ..........................................................................41

Chapter 4 Optimization .............................................................................43
  4.1 Penalty Function Definition .....................................................................45
    4.1.1 Least Squares Minimization ............................................................45
    4.1.2 Assignment of Weighting Factors ......................................................46
    4.1.3 A Semiconductor Device Parameter Penalty Function .....................47
  4.2 Parameter Constraints ...........................................................................48
  4.3 Optimization Methods ...........................................................................49
    4.3.1 Downhill Simplex Method ...............................................................49
    4.3.2 Gradient-Based ................................................................................50
    4.3.3 Simulated Annealing .........................................................................51
    4.3.4 Exhaustive Search ............................................................................52
  4.4 Incorporation into System ......................................................................53

Chapter 5 Software Infrastructure ............................................................54
  5.1 Tool Hierarchy .......................................................................................55
  5.2 Implementation Issues ...........................................................................57
    5.2.1 Applications vs. Applets ....................................................................57
    5.2.2 Other Implementation Information ....................................................59
  5.3 Important Classes & Packages ...............................................................60
    5.3.1 Main Object - DSETmainFrame .......................................................60
    5.3.2 Exploration Engine Object - DSETEngine .......................................61
    5.3.3 Parameter Representation - GenericParam and ParamSet Objects ....63
      5.3.3.1 GenericParam ..............................................................................63
      5.3.3.2 ParamSet ....................................................................................66
    5.3.4 Model Objects, the DeviceModel interface .......................................68
    5.3.5 The Model Object Library Manager - How to Add Models ...............72
    5.3.6 Optimizer Module & Relevant Classes ..............................................73
      5.3.6.1 Optimizer class ............................................................................73
      5.3.6.2 MultivariableFunction interface ....................................................75
      5.3.6.3 ModelPenaltyFunction class .........................................................76
      5.3.6.4 GradientFunction interface ............................................................77
      5.3.6.5 CalcGradientFunction class ..........................................................77
      5.3.6.6 OptimizationInfo class .................................................................78
Chapter 6  Test Scenarios .................................................................83
6.1  Case 1: Selection of Device Structure.................................83
6.2  Case 2: Sensitivity Analysis .................................................87
6.3  Case 3: Curve Fitting & Parameter Extraction .....................89
6.4  Case 4: Present and Future Technology Exploration .............91

Chapter 7  Conclusions .................................................................95
7.1  Future Research Work .............................................................95
7.2  Incorporation into the CAD environment ...............................96

References ................................................................................98

Appendices:
App. A:  Optimization Algorithms .............................................100
App. B:  Model Derivations .........................................................107
List of Figures

FIGURE 1. Flowchart of current selection procedure for semiconductor device parameters ................................................................. 14
FIGURE 2. Flowchart of feasible parameter set generation using proposed tool ...... 15
FIGURE 3. Block Diagram of Device Parameter Space Exploration Tool .............. 20
FIGURE 4. Parameter adjustment criteria ................................................................. 24
FIGURE 5. Flowchart of exploration engine’s parameter space solution algorithm .......................................................................................... 26
FIGURE 6. Block diagram of operation of system ................................................ 29
FIGURE 7. Cross section of a bulk NMOS device .................................................. 31
FIGURE 8. Cross section of a SOIAS device .......................................................... 34
FIGURE 9. Cross section of a Silicon-Germanium device layer structure ............... 36
FIGURE 10. Searching along a curve vs. “around” a curve ...................................... 44
FIGURE 11. Package hierarchy for Device Parameter Space Exploration Tool ....... 55
FIGURE 12. Examples of a Java interface and its use ............................................. 56
FIGURE 13. Sample model file template ................................................................. 70
FIGURE 14. Applet View ....................................................................................... 80
FIGURE 15. Graphical User Interface for Exploration Tool .................................. 81
FIGURE 16. Tool Application/Applet hierarchy ................................................... 82
FIGURE 17. MATLAB plot of Channel Width-Oxide Thickness combinations .......... 86
FIGURE 17. Parameter Space Exploration for Parameter Extraction ..................... 89
List of Equation Sets

Equation Set 1. Model for a simple bulk CMOS device. ........................................32
Equation Set 2. Bulk CMOS Charge-sheet based model ......................................33
Equation Set 3. Silicon-On-Insulator with Active Substrate model ..................35
Equation Set 4. Oxide thickness modification for Si-Ge model.........................36
List of Tables

TABLE 1. Feasible Parameter Combinations for a bulk CMOS device ..................84
TABLE 2. Comparison of tox-W combinations for bulk vs. SOIAS .....................85
TABLE 3. Current Sensitivity to Variations in Oxide Thickness..........................87
TABLE 4. I-V Data Points Generated from Charge-Sheet Model ..........................88
TABLE 5. Substrate doping-oxide thickness combinations for bulk CMOS and Si-Ge .................................................................90
TABLE 6. Electrical Data used for Formulated Penalty Function ..........................92
TABLE 7. Optimization Results using Downhill Simplex Method ........................92
TABLE 8. Optimization Routine Performance Results for Sample Function ..........106
Chapter 1

Introduction

One of the most important decisions a circuit or device designer must make is the selection of a specific device structure which is to be used within a given circuit design. To find desirable device characteristics, a designer must often revert to potentially time consuming techniques and calculations (e.g., numerical device simulators such as MEDICI [TMA]). This thesis describes the development and implementation of a tool which allows a designer to rapidly explore the device parameter space for a given technology, as well as across different technologies, and which is of assistance to circuit and device designers in determining the optimal device characteristics to use in a given circuit.

1.1 Device Parameter Space Exploration

In general, any semiconductor device can be described by a device model. Device models are sets of equations which define the current-voltage characteristics of a particular device as a function of several parameters, some of which may represent environmental effects on a device (such as temperature), some of which may represent physical properties of the device (such as channel width), and some of which may represent fitting parameters (such as the channel length modulation parameter). These parameters and the sets of their feasible values form a parameter space which can be thought of as the set of all possible combinations of semiconductor device parameter values.
The idea of semiconductor device parameter space exploration is to examine the subset, or space, of semiconductor parameters under defined electrical constraints, and to determine a method of describing and utilizing the subsets to provide useful information. Specifically, the object of the tool is to determine the feasible parameter space of a specific device model given user-defined electrical and parameter constraints. Optionally, it is also possible, through formulation of a penalty function, to perform optimization routines to determine specific sets of parameters which may be of interest.

### 1.1.1 Semiconductor Device Parameter Classifications

As described before, semiconductor device parameters can be classified as either environmental, physical, or fitting parameters. The classification of most interest is the set of physical parameters, simply because it is the set of parameters the designer has the most control over. The set of physical parameters can be further separated into two classifications: layout parameters and process parameters.

Layout parameters, as the name implies, are parameters which are selected by an engineer during the layout phase of circuit design, usually to meet a certain electrical or performance constraint. Process parameters are physical parameters which are inherent in the fabrication of the device, and are usually shared by all devices made by the particular process (within some nominal variation). For instance, the two layout parameters most common in VLSI design using bulk CMOS devices are channel length and width of a given device. In general, layout parameters are set by the circuit designer on a device by device basis. On the other hand, process parameters such as substrate doping and oxide thickness are often fixed for the entire set of transistors by whichever semiconductor process the circuit designer selects from the available processes, or the device designer chooses to implement for a given circuit or circuit technology.

### 1.2 Definition of Problem

Given a device model which defines a current-voltage relationship as a function of a vector of semiconductor device parameters and a voltage vector (in general, this vector can be thought of as a set of voltages applied to the terminals of a device), there exists a
parameter space of all physical parameters of the device. It is unwise to explore variations in fitting parameter space since such parameters will be fixed in the circuit model for most applications. The goal of exploring parameter space is to find a region within the space of all physical parameters where the parameter values of every point within the region meets some specific electrical criteria. The specific electrical criteria which is used to generate parameter spaces is current output for a given voltage.

It is also possible, through a definition of a penalty function, to use optimization techniques to explore the parameter space and determine the optimal set of parameter values within a given parameter space.

Mathematically put, given:

(1) \( P \), the space of all physical parameters of a device,

(2) a current-voltage set \([I_0, v_0]\) (where \( I_0 \) is a fixed current and \( v_0 \) is a set of voltage values), and

(3) a device model which defines the relationship \( I=F(p,v) \) (where \( p \) represents a set of parameter values, and \( I \) represents current),

finding a parameter space is equivalent to finding:

\[
\{ p \in P \mid F(p,v_0) = I_0 \} \quad (EQ \ 1)
\]

1.3 Motivation

One of the problems facing circuit designers is the selection of feasible device characteristics for a given circuit. While circuit designers can simply run numerical simulations such as MEDICI, or circuit simulators such as SPICE, to verify the feasibility of particular circuit model parameters, this involves selection of the parameters beforehand and iterative simulation until the desired parameters are reached. Often situations will occur where a circuit designer simply selects a particular set of parameters based on the
current technology available. If a particular electrical specification is not met, the designer resorts to reworking the configuration of the circuit until that specification is met. This may involve adding extra devices to the circuit or changing parameters which the designer would rather remain fixed.

Most circuit designers only modify layout parameters and choose from a finite set of foundry processes. Some circuit designers also help specify requirements to a device design group; these designers would benefit greatly from a tool which can be used to determine feasible device structures for a given circuit.

The usefulness of an exploration tool is due to its implementation as a rapid analytical modeling approach to solving for semiconductor device parameter space. Thus, instead of requiring the designer to input a large set of fully specified parameter constraints and checking to see if the electrical criteria are met (a very time-consuming process), this tool would allow the user to specify the electrical criteria once, and a range of values for specific parameters, and receive specific device parameter space information for the desired parameters.

The current procedure for selection of semiconductor device parameters performed by a circuit designer is described in Figure 1. The circuit designer selects a single process for fabrication of all devices and certain layout parameters for each transistor (or type of transistor) in a given circuit. After circuit simulation using a tool such as SPICE, each transistor is checked to ensure that it meets the desired electrical specifications. If it does not, then the layout parameters are altered. If the circuit is such that the layout parameters for different devices cannot be reasonably chosen to satisfy all of the electrical constraints, then a different fabrication process or even a different technology must be chosen and the procedure must be repeated until a suitable technology, process, and layout are obtained.

This procedure can be quite time consuming since each iteration requires a full circuit simulation. While an experienced circuit designer may be able to reduce the number of iterations by using knowledge about a particular circuit to adjust layout parameters sufficiently, at times a shorter decision time can be desirable, especially if only a rough guess for the solution is required.
The procedure outlined in Figure 2 describes the proposed method for feasible semiconductor device parameter selection. The new procedure seeks to determine the feasible device parameter sets from predefined parameter constraints and specified electrical criteria. The idea is that exploring the semiconductor device parameter space for multiple devices may lead to a specific process parameter set which can be used for all devices, and layout parameters which correspond to that process parameter set. In the new procedure, parameter constraints are specified, which apply to all devices. These constraints may be determined by either feasibility of manufacturing or availability of particular fabrication processes. Then, electrical specifications are selected for each transistor in the circuit. The feasible parameter space is computed to yield a set of feasible parameter value combina-
tions. The process is repeated for each transistor in the circuit. These generated parameter sets represent feasible parameter value selections for devices. The parameter sets are then analyzed to find sets which have similar process parameter values. These values can be used to decide the particular process to use for the specified circuit. The result are sets of feasible parameters for each device in the circuit.

FIGURE 2. Flowchart of feasible parameter set generation using proposed tool

The feasible parameter sets generated can then be searched for an optimal set of parameters, based upon some specified criteria (minimization of a user-programmed pen-
alty function). Alternatively, the option is also available to explore the parameter space about a particular parametric curve (generated for a particular electrical criteria) for a possible optimal parameter set (to minimize the aforementioned penalty function).

1.4 Device-Parameter Space Exploration Tool (DSET)

To implement the technique of semiconductor device parameter space exploration using rapid analytical modeling, it is necessary to create a tool which can be used to define the electrical specifications for devices and calculate the feasible parameter space for a given device model. Such a tool could be used as an alternative means for selecting device parameters for circuits.

The tool, named DSET, was created as part of research described in this thesis. It was implemented in Java as both an applet and an application program. Details about the software implementation are found in Chapter 5.

1.5 Applications

There are a wide variety of scenarios where exploration of semiconductor device parameter space is useful to a circuit designer in terms of providing extra information not normally available, or providing information faster than current methods allow. These example scenarios are briefly described below and are described in further detail in the case studies of Chapter 6.

1.5.1 Feasible Device Parameter Generation

In this scenario, a circuit designer knows the electrical specifications for each device in a given circuit, but might want to explore some or all of the layout or process parameter dependencies in the circuit. Using this tool, it would be possible for the circuit designer to determine how much channel width could be altered by selection of a different substrate doping. During the redesign process, a circuit designer could then use this information to decide whether to modify the layout parameters, or select a new technology process which would allow for less modifications to the existing layout parameters.
1.5.2 Exploration of Future Technologies

In the research and development stages of a new type of technology, a model for the device’s behavior may be readily available while the actual fabricated devices may not exist. (Or perhaps they are not economically feasible to manufacture.) A feature of the tool described in this thesis allows the user to add the model to this tool’s existing set of models and evaluate its feasible parameter space to determine whether or not it would be reasonable to construct a given circuit using that particular technology. It would also provide a rough guess as to how much alteration within the circuit would be required if that particular technology was used in fabrication, and could be used as a basis for a decision to pursue further research and development of such technology.

1.5.3 Sensitivity Analysis

Since the purpose of this tool is to explore semiconductor device parameter space, it also can be used to determine parameter dependencies, and the sensitivity of one parameter with respect to another. Process parameter variations always occur when fabricating a given circuit, so the parameter sensitivities are crucial in determining the overall performance of the device within the circuit.

Specifically, this exploration tool can answer the question: if a specific process parameter (e.g. substrate doping) varies by a nominal amount, how much will it affect the electrical characteristics? Another question which could be answered is: if a specific process parameter varies by a nominal amount (such a change may be estimated from previous fabrication runs), by how much is it necessary to vary a given layout parameter (or parameters) to meet a given electrical specification?

1.5.4 Technology Comparisons

A circuit designer can also use this tool to determine which of the available technologies is best suited for a given circuit design. If a circuit designer has a given electrical specification and given process parameter constraints, this tool can be used to generate the feasible device parameter spaces for each of several specific technologies (e.g., bulk CMOS, Si-Ge, or SOIAS). Using the information provided by these generated spaces, the
circuit designer can select the technology which fits into the given process parameter constraints. This tool can be used to compare parameter variations and dependencies both within technology families (0.5 μm vs. 0.2 μm bulk CMOS), and across different technology families (bulk CMOS vs. Si-Ge).

1.6 Thesis Overview

Chapter 1 of this thesis serves as an introduction to the reader to the concept and applications involving the semiconductor device parameter exploration tool. Chapter 2 gives a system overview and describes, on an abstract level, the various components of the system and the system operation. Chapter 3 describes the device models which were used with this tool, and explains some of the issues involved in the implementation of such models; detailed descriptions of these device models are presented in Appendix B. Chapter 4 gives a description of optimization methods which are used within the tool. The implementation of the tool in Java is described in Chapter 5, which also documents some of the important classes and methods. Chapter 6 describes the case studies which have been tried with this tool, and offers some other potential test scenarios for its use. Chapter 7 provides a conclusion to the research work, and offers suggestions for future areas of development.
Chapter 2

System Design

The system inputs (which may be user-defined or default values) are: a set of electrical criteria, sets of parameter constraints and, optionally, a penalty function which would be used by the optimizer. The output of the tool is the feasible device parameter space for the given inputs, and, if optimization was desired, the optimal set of parameters for the specified penalty function.

The system is meant to analyze the static electrical behavior of a device in its parameter space exploration. Although the specific applications of this tool which were studied analyzed four-terminal semiconductor devices, the system could be extended to analyze any type of “black box” device which had one major output value, provided that the proper device model object protocol was followed and the system infrastructure was modified to include a procedure to analyze such data. Extensions to analyze dynamic behavior of a device may require modification of the object protocol to allow for incorporation of additional information, and modification of the system infrastructure to accept and properly analyze the information.

The system abstraction consists of three distinct groups, each of which consists of several blocks. The overall system block diagram is shown in Figure 3.
2.1 System Components

The system consists of a main module whose primary job is to control the other modules, a set of model objects, an exploration engine, an optimization module, and an output display. These modules, acting together, provide the information about the feasible device parameter space.
2.1.1 Main

The main module operates as the overseer of the entire tool. Its primary job is to arbitrate and control passage of information between the other modules of the system. It also performs some small bookkeeping and other management tasks to insure the proper interaction between the other modules.

2.1.2 Model Objects

The idea of a device model object is an encapsulated software object which implements a specific device model. The specific encapsulation includes a generic interface to insure that different device model objects can be interchanged within the tool as simply as possible. In addition, the interface between the device model objects and the exploration engine is generic enough that the design and implementation of the exploration engine are kept separate from the implementations of the device model objects.

The device model object incorporates the device model itself (e.g. the set of equations defining the electrical behavior of a given device), the current value of all parameters, bookkeeping information, and other important information which may be extracted from the model using the proper interface. In this way properties of a given model can be embedded into the model object which represents that model and can be retrieved when necessary or desired.

There is a standard generic model object protocol which must be followed by any model which is to be included into this system. Having a generic model object protocol allows for generic calls to model objects (for calculations of currents and other model functions) which can be made by the rest of the system (e.g. the exploration engine) and which will apply to any model object included in the system. It also allows the development and implementation of the model object to be kept separate from the development of other modules within the system. The model object protocol is specified in Chapter 3.

2.1.3 Model Library Manager

This module is responsible for loading and keeping track of the individual model
objects within the system. It also is responsible for providing a set of valid model objects to the main module for use with the exploration engine and optimization tool. The implementation of this module must be kept separate from the implementation of the model objects such that additional model objects can be developed and added to this system without modifying this module. Since all model objects implement a defined protocol, it is sufficient for the Model Library Manager to insure that all model objects allowed to be passed into the system comply with the specified protocol.

2.1.4 User Input Module

This module operates as an interface between the user and the tool. It is responsible for requesting and processing user inputs and passing them to the main module for use. It is in this module that all user interaction is performed, including parameter constraint specification and electrical criteria selection, as well as other side tasks such as requests for saving and loading data, and output display adjustments.

2.1.5 Exploration Engine

Device models describe how the parameter space maps to the electrical performance space. However, for this tool, the inputs will be electrical performance criteria, and the outputs parameter spaces. Therefore, a method must be developed whereby it is possible to perform parameter extraction. In the context of this section, the term “parameter extraction” refers to the derivation of a specific set of parameters which map to desired electrical criteria, given a set of parameter constraints.

The goal of the exploration engine is to solve for a specific parameter space, given a set of electrical criteria. A parameter space can be thought of as a group of parameter sets, where parameter sets are specific combinations of parameter values. Thus, it is necessary to formulate a method to solve the parameter extraction problem which occurs naturally from solving for a specific parameter space. It is important to note that although any parameter extraction technique or algorithm may be implemented for use with the exploration engine, the performance of the algorithm determines the speed at which the exploration engine can calculate the desired parameter space. Thus, efficient algorithm
performance is desired in order to reduce computation time.

The method of parameter extraction involves an iterative dynamic step-climbing algorithm which provides an accurate estimate to the actual value of a given parameter. The algorithm first fixes all but one given parameter, selects a default value for that parameter, calculates the performance output, and checks if that output meets some desired error tolerance. If not, the parameter is adjusted by the algorithm, and another iteration is performed. The process is repeated until the error tolerance is met. A nominal error tolerance of 0.01% of the desired current (for a given voltage bias) is used as the metric for ending the algorithm. Such an error tolerance leads to a reasonable estimate for device parameters without a very large algorithm convergence time.

The specific method of parameter adjustment is a dynamic adjustment approach meant to minimize the error between the calculated current and the desired current for the given parameter. The adjustment algorithm assumes concavity of the error function with a single global minimum. This is true if it is assumed that if device model current is expressed as a function of any one parameter (and other parameters are kept constant), then the current function is monotonic, and only one parameter value will lead to any given current value (which is not the case in general if current is expressed as a multivariable function, since different parameter value combinations can possibly lead to the same current).

First, using a default parameter value as a starting point, a range of parameter values is determined about that point. At each point, a corresponding value of absolute difference between desired and calculated current is formed. If an error value corresponding to a point in the scan range is within the error tolerance, than that parameter value is selected as the best estimate of the parameter value which meets the given electrical criteria, and the adjustment algorithm terminates and returns the corresponding parameter value to the exploration engine. If a desired error tolerance is not found, the scan range adjusts itself in both position along the parameter axis and in overall size, to best determine where a minimal error value might lie. If the error values in the scan range are monotonic in one direction, then the range is adjusted such that the center point of the new scan range of parameter values is the parameter value corresponding to the smallest error term. The size
of the scan range is also doubled, although the number of points remains fixed (i.e. the step size between points is doubled). If a minimum is found to be within a scan range of values, then the range is also adjusted such that the center point of the new scan range of parameter values is the “smallest error term” parameter value, but now the scan range size is halved. This allows for swift convergence to the global minimum. The parameter adjustment criteria is shown in Figure 4.

Each iteration produces a set of error values corresponding to points along a parameter space.

minimum at $S_b$: $S_{c, next} = S_b$ double step size
minimum at $S_e$: $S_{c, next} = S_e$ double step size
minimum at $S_o$: $S_{c, next} = S_o$ halve step size

**FIGURE 4.** Parameter adjustment criteria

It is important to note that this one-dimensional minimization of error approach only works because of the assumed concavity and presence of a single global minimum of the given function, and should not be used as a solution for general minimization problems.

The technique of parameter extraction is embedded into the exploration engine. The engine then can use the parameter extraction technique in a parameter space-walking algorithm, shown in Figure 5. First, the user specifies which parameters should be varied and which parameters should be fixed. It is assumed that in most cases the user will only desire to vary a few parameters at a time; if a large number of parameters is varied, the designer may have difficulty discerning the direct trade-offs among parameters. In addi-
tion, the more parameters that are varied, the larger the parameter space the exploration engine must walk, and the longer the computation will take. The user also should specify ranges of interest for each parameter to be swept across, and step sizes for each swept parameter range; the user should also specify a technology to be explored by selecting a specific device model to use with the parameter extraction algorithm.

The basic algorithm is as follows: the designer will specify some set of electrical criteria (e.g. current drive for a certain voltage bias) for the devices, and will select N (N must be greater than 1) parameters, (N-1) of which are varied across specified ranges. The other device parameters will be set at specific values (default values are available). The exploration engine will then sweep/step across the (N-1) ranged parameters and calculate the unknown parameter at each step. The output data will be passed back to the main module, where it can be further processed if necessary. This method can be used to calculate any number of parameter variation responses; however, for purposes of outputting information to the users, it is anticipated that only parameter spaces of two or three variables will be of any use for visual inspection. Higher order parameter spaces may still be useful for optimization routines and other computer processing.

For example, if a designer has some specified current and voltage criteria for a bulk CMOS device, and wanted to explore how the modification of one parameter (e.g. channel length) would affect another parameter (e.g. oxide thickness) while keeping the other parameters constant, this tool could be used by specifying the electrical criteria as input, the channel length and oxide thickness as variable parameters, and the channel length as the parameter to be swept across. If desired, the user can put a constraint on the swept variable (e.g. channel length less than 0.1 microns), select a step size, and also specify bulk CMOS as the technology.

2.1.6 Optimizer

Since the output from the exploration engine will be data corresponding to a feasible parameter space for a given device, it is possible to use this data to find the optimal device characteristics. One method of optimization involves examining the output generated by the exploration engine and, using some user-specified (defaults available) set of
FIGURE 5. Flowchart of exploration engine’s parameter space solution algorithm

constraints, and user-defined (default available) optimization function, execute an optimization search which selects the best set of device characteristics to meet the user’s specifications. Another type of optimization can be used to explore the region about a particular parametric curve which corresponds to an exact match to an electrical criteria. Several other optimization routines can be used for this purpose. The exact details of the optimization algorithms are presented in Chapter 4.

2.1.7 Output Display

The data which is calculated by the exploration engine must be presented to the user in a useful format in order to be of any assistance. Graphical output capability to dis-
play a two-dimensional parameter space has been implemented as a way of providing the user with a direct plot of parameter variations to visually inspect for regions of interest. In this way, by direct examination, the user can immediately see how two specified device parameters vary with respect to each other and can visually select a region where the optimal set of parameters exists. The curves serve to highlight the feasible device parameter space to the user, which can then be utilized in the selection of device parameters for a given circuit.

In addition, provisions have also been made to allow for file-based output, such that the required data can be saved and retrieved at a later time, for comparisons or future computer processing. Such file-based records have a specified table-based format which allows for easy retrieval by this tool, or by any program which can be configured to read text-based files.

2.2 System Operation

The operation of this tool can be broken down into several steps, shown in Figure 6. During the Start-up and Initialization phase, the Model Object Library Manager loads and initializes copies of all model objects to be available to the tool. Initialization of a model typically involves resetting all parameter values to their defaults. Next, the Library Manager sets all of the fitting parameters within each of their models to desired values (or keeps the default values). The Library Manager then passes the copies of the models to the main module. At the end of the initialization phase, the tool is ready to accept user inputs and generate feasible parameter spaces.

The next step is User Input Acquisition. This includes specification of the technology and electrical criteria, selection of values for fixed parameters, selection of ranges for variable parameters, step sizes for those ranges, and optimization information (parameter constraints and optimization function). This information is requested by the User Input module and passed to the main module.

The next step is Feasible Parameter Space Generation. The main module then passes the information to the exploration engine and requests a feasible parameter space.
The exploration engine utilizes the parameter extraction technique to generate a feasible parameter space and passes this information back to the main module. The main module then has the option of displaying the feasible parameter space to the user via the Output Display module, or can pass the feasible parameter space to the optimization routine. This choice is made depending on the user’s selection during the user inputs acquisition phase.

If optimization is desired by the user, the next step is Optimization. The optimizer receives the feasible parameter sets and searches those sets while minimizing a user-specified penalty function (also referred to as optimization function in this section). Upon finding the set of parameter values which minimizes the given penalty function, the optimizer passes that set of parameters to the main module, which can then display the output to the user via the Output Display module. This process may be repeated as desired by the user, each time specifying different inputs to produce different feasible parameter spaces or different sets of optimal parameters.

Alternatively, if the user decides to explore the space around a feasible parameter set, then the user also has the option of bypassing the feasible parameter set generation and invoking optimization directly. In this way the parameter space about parameter values which meet a specific electrical criteria may be explored.
FIGURE 6. Block diagram of operation of system
Chapter 3

Device Models

As specified before, a device model is a set of equations which describe the electrical behavior of a particular semiconductor device as a function of a set of parameters inherent to that device. Typically, such models are incorporated into device-level (e.g., MEDICI) or circuit-level (e.g., SPICE) simulators for use in circuit design and analysis. Each device technology generally requires a different device model, since electrical behavior as a function of device parameters varies from device to device; the type and number of parameters inherent to a given device model will also vary from model to model.

There may also exist multiple distinct models corresponding to the same device technology. Each of these models serves to provide the designer with a varying degree of accuracy for a given device. Often such “higher level” device models will incorporate many high-order effects which may be of concern to a circuit designer for a given circuit. The penalty paid is extra computation time to factor in such higher order effects on circuit performance. Another disadvantage of higher order models is that such models often require selection of fitting parameters which must be extracted from real device data or device-level simulators. Such extraction may be time consuming and difficult to perform accurately.
3.1 Device Models

Three types of device models have been implemented and used in the exploration tool. Each device has its own unique characteristics and advantages for use in a given circuit.

3.1.1 Bulk CMOS

CMOS (Complementary Metal-Oxide-Semiconductor) is the technology used to fabricate most of today's VLSI circuits. CMOS circuits contain both NMOS and PMOS devices. An NMOS device consists of a p-type substrate with two n-doped regions implanted on opposite sides of a polysilicon gate, insulated from the substrate by silicon dioxide. There are four terminals on the device: gate, source, drain, and bulk. The bulk of the devices is almost always grounded or fixed at the lowest potential in the circuit (for NMOS).

![Cross section of a bulk NMOS device](image)

FIGURE 7. Cross section of a bulk NMOS device

The advantage of using bulk CMOS devices in a given circuit is that it is a well known technology whose behavior can be modeled very accurately in contemporary circuit simulators. In general, field-effect transistors are preferred over the other major classification of transistors (bipolar junction transistors, or BJTs) because they consume less power in a circuit and have higher input resistances.

There is a vast quantity of models which represent bulk CMOS devices, with the main difference between the models occurring due to differences in the amount of higher order effects included within the models. Two models have been chosen for use in this
tool: a region-based CMOS model which is the classic model found in any undergraduate level electronics text, and a surface potential based model which is more complicated and incorporates a few higher order effects.

Region Based

The region based model is the simplest model which is used in this tool. It incorporates the classic region-based equations found in any electronics textbook dealing with MOS devices [TSIVIDIS]. There are separate equations for the three regions (cutoff, triode, and saturation) of device behavior. Such regions are defined by electrical quantities applied to the terminals of the device.

\[
I_{ds} = \begin{cases} 
0 & \text{if } V_{gs} < V_{th} \\
\frac{1}{2} \mu C_{ox} \frac{W}{L} \left( 2(V_{gs} - V_{th})V_{ds} - V_{ds}^2 \right) & \text{if } V_{ds} \leq V_{gs} - V_{th} \\
\frac{1}{2} \mu C_{ox} \frac{W}{L} (V_{gs} - V_{th})^2 & \text{if } V_{ds} > V_{gs} - V_{th} 
\end{cases}
\]

\[
V_{th} = V_{T0} + \gamma \sqrt{\frac{2 \phi_F - V_{bs} - \sqrt{2 \phi_F}}{V_{bs}}}
\]

Equation Set 1: Model for a simple bulk CMOS device

In Equation Set 1, \( \mu \) is the carrier mobility, \( C_{ox} \) is the oxide capacitance per unit area, \( W \) and \( L \) are the channel width and length, \( \gamma \) is the body effect coefficient, and \( \phi_F \) is the Fermi potential, and \( V_{T0} \) is the nominal threshold voltage of the device. \( V_{gs}, V_{ds}, \) and \( V_{bs} \) are the gate-to-source, drain-to-source, and bulk-to-source voltages, respectively.

The advantages of this model are that it is easy to compute and provides a reasonable estimate of actual device behavior. It works well for simulations involving large-scale devices which do not operate in the subthreshold regime, or for initial simulations where all that is required is a rough estimate of electrical behavior. It also contains no fitting parameters and thus does not require acquisition of values of such parameters.

This is not the type of model to use if extremely accurate modeling of device behavior is desired, since it neglects to include any higher order effects which occur in real
devices. It also neglects to include the diffusion component of current, thus resulting in a poor approximation of weak inversion and subthreshold regions of operation.

**Surface Potential Based (Charge Sheet) Model**

The charge sheet model is a more accurate model of bulk MOS transistor behavior. It includes several second-order effects, as well as fitting parameters to allow accurate modeling of a real device by fitting to the desired real device data. For this model, there is one equation which is valid for all regions of operation.

\[
I_{ds} = \frac{C_{ox} W_{eff} \mu_s}{L_{eff}} [V_0 - \frac{\alpha V_{ds}}{2}] V_{dsx}
\]

where

\[
h = \frac{\left( \frac{\mu_s}{v_{sat}} \right)^2 V_{dsx}}{1.5 + \frac{\mu_s}{v_{sat}} V_{dsx} L_{eff}}
\]

\[
V_{dsx} = V_{dsat} \left( 1 - \frac{\ln \left( 1 + \exp \left( 10 \left( 1 - \frac{V_{ds}}{V_{dsat}} \right) \right) \right)}{\ln(1 + \exp(10))} \right)
\]

\[
\alpha = 1 + \gamma \delta
\]

\[
\delta = \frac{1}{2} \left( \frac{\phi_{ss} - \phi_t}{\phi_{ss} - \phi_t} \right) \left( 1 - \frac{1}{1.43 + 1.41(\phi_{ss} - \phi_t)} \right)
\]

\[
V_0 = V_{gb} - V_{fb} - \phi_{ss} - \gamma \phi_{ss} - \phi_t + \alpha \phi_t
\]

**Equation Set 2: Bulk CMOS Charge-sheet based model**

In Equation Set 2 above, \( W_{eff} \) and \( L_{eff} \) are the effective channel width and length, \( v_{sat} \) is the saturation velocity, \( \phi_t \) is the thermal voltage, \( V_{fb} \) is the flatband voltage, \( C_{ox} \) is the oxide capacitance per unit area, \( \gamma \) is the body effect coefficient, and \( \mu_s \) is the carrier mobility. The model shown in Equation Set 2 omits the equations for \( V_{dsat} \), the saturation voltage, and \( \phi_{ss} \), the surface potential on the source side, because these quantities are solved using iterative procedures which are explained in further detail in Appendix B.
The model is derived using the charge-sheet analysis of an MOS transistor. The expression achieved for the drain current, which takes into account both the drift and diffusion components and also mobility degradation effects, holds in the strong and weak inversion and saturation regions of operation, and results in a continuous function of all bias voltages. This approach has been shown to predict results which satisfactorily compare with experimental data [TURCHETTI]. The model is derived by using a 2-D Poisson’s equation with Gauss’ Law to derive an expression for the surface potential, and using the surface potential in a modified form of the one-dimensional current continuity equation for electrons. The full model derivation is given in Appendix B.

The important features of the model are that it is physical, scalable, and compact. It will model device behavior accurately once fit to real device data, and has a reasonable simulation time when implemented in computer code. It includes several second-order effects and can be altered to allow for more higher order effects if desired. It also has only a small number of fitting parameters, so fitting and calibration time is minimized. The model was fitted to MEDICI IV data to select default values for all fitting parameters.

3.1.2 Silicon-on-Insulator with Active Substrate (SOIAS)

SOIAS is an emerging technology which belongs to the family of Silicon-on-Insulator (SOI) devices [YANG]. SOI transistors are four terminal devices, structured similarly to bulk MOS transistors, except with a layer of oxide serving as the substrate. SOIAS devices are SOI transistors with an isolated backgate which can be used to control the threshold voltage via the body effect.

![FIGURE 8. Cross section of a SOIAS device](image)
There are several advantages of using this type of technology in a VLSI system. First, SOI technology offers the benefit of reduced parasitic capacitances from drain to bulk and source to bulk, as well as the advantage of improved isolation, allowing for greater density of devices on a given chip. SOIAS devices have the further benefit of the ability to control threshold voltage, which is of great use in certain low-power applications, where the desired device should have a high threshold voltage when not active (to allow for low leakage currents), but a low threshold voltage when active (to maximize current drive).

The SOIAS model is a surface-potential based model that is physical, scalable, and compact. Derived by Narendra [NAREN], it strives to minimize the number of fitting parameters required. It was fitted to simulated (MEDICI) IV data to select default values for all fitting parameters. The exact details of the derivation of the model can be found in Appendix B. Equation Set 3 shows the final current equation.

\[
I_{ds} = \frac{\mu_s C_{ox} W}{L_{eff} + h V_{dsx}} \left[ V'_{gs} + \frac{t_{ox}}{t_{box}} (V'_{bs} - \phi_b) - \frac{q N_{a_{tsi}}}{C'_{ox}} + \phi_i - \frac{1}{2} (V_{dsx} + 2 \phi_{ss}) \right] V_{dsx}
\]

\[
V'_{gs} = V_{gs} - V_{fbf}
\]

\[
V'_{bs} = V_{bs} - V_{fbb}
\]

**Equation Set 3: Silicon-On-Insulator with Active Substrate model**

In Equation Set 3, \(t_{ox}\) and \(t_{box}\) represent the oxide thicknesses at the front and back gates, \(t_{si}\) is the thickness of the silicon film, \(L_{eff}\) is the effective channel length, \(V_{fbf}\) and \(V_{fbb}\) represent the flatband voltage at the front and back gates, the equation for \(h\) is the same as in Equation Set 2, and \(V_{dsx}\) represents a voltage smoothing function similar to the function shown in Equation Set 2. Both \(V_{dsx}\) and the surface potentials at the source and back gate, \(\phi_{ss}\) and \(\phi_b\), are calculated using procedures explained in Appendix B.

The model is derived using an approach similar to the charge-sheet analysis used to derive the model of Equation Set 2, modified to account for the effect of the surface potential at the back oxide interface (which does not exist in bulk CMOS devices).
3.1.3 Silicon-Germanium (Si-Ge)

Silicon-Germanium devices are also an innovative technology that looks to be important in future VLSI circuits [SADEK]. Figure 9 displays a typical example of the structure of a buried-channel silicon-germanium device. Si-Ge devices have a silicon-germanium channel buried within the device, connecting the source and drain regions. There are four terminals, just as in bulk CMOS and SOI devices: gate, drain, source, and bulk.

The advantage of using Si-Ge devices occurs due to the higher electron and hole mobilities present in such devices. The larger current drive due to the higher mobilities results in faster performance for the same voltage biases, without requiring modifications to layout or other parameters.

Modeling of a Si-Ge device is achieved by modifying a CMOS bulk model in the following ways: first, the mobility values $\mu_s$ are increased to reflect the higher mobilities present in Si-Ge devices; second, the oxide thickness value is changed to incorporate the effects of the silicon cap layer, according to the following equation:

$$t_{ox}' = t_{ox} + \frac{e_{ox}}{e_{si}}t_{si}$$

\textbf{Equation Set 4: Oxide thickness modification for Si-Ge model}

where $t_{ox}$ is the oxide thickness, $t_{si}$ is the silicon film thickness, and $e_{ox}$ and $e_{si}$ are the permittivities of oxide and silicon, respectively.
The modeling of silicon-germanium devices is an ongoing research topic [SALAMA], and the model shown here is one of the simplest to implement. More complex models can be implemented to incorporate such higher-order effects like velocity overshoot and surface inversion.

The model equations are equivalent to those for the bulk CMOS charge-sheet model except for the modifications mentioned. More details on the derivation are found in Appendix B.

### 3.2 Calibration and Fitting Issues

In order to achieve a model which accurately mimics a real device, it is usually necessary to include fitting parameters into the equations for the electrical characteristics for the device. Fitting parameters have no corresponding physical value, but instead serve to allow the model to be calibrated to data obtained from actual device electrical measurements, or a device-level numerical simulator such as MEDICI.

While even uncalibrated or partially calibrated data may be useful for some purposes (e.g. educational or initial "rough guesses"), for the most accurate results, calibration of the model is essential. The device model has several fitting parameters whose default value must be found. These fitting parameters may be modified by the user during operation of the tool, but the default values initially present within the model should allow for a reasonably accurate estimation of real device data.

If a user of the tool has specific device data which reflects desired model electrical behavior, the user simply changes the values of the default fitting parameters to reflect the desired fitting parameters corresponding to the specific device data. The present tool is not structured to perform fitting parameter extraction given a set of device data.

All models used in this tool, unless noted otherwise, were manually fitted to MEDICI simulated electrical characteristics to determine default values for all fitting parameters. Since it is nearly impossible to adjust fitting parameters such that the model-generated electrical characteristics exactly match the MEDICI-generated data, the fitting
parameters for the model were selected such that the fit was within a nominal error tolerance (<10%) using an RMS error formula typically used for parameter extraction [WARD].

3.3 Model Objects - Protocol

The encapsulated model object includes a generic protocol which must be followed by any model to be incorporated into this tool. The protocol defines a list of certain characteristics that all model objects must have. A description of each characteristic follows, with the region-based bulk CMOS model used as an example to illustrate the corresponding characteristic within this simple model. The computer code implementation of the protocol is described in Chapter 5, as well as implementations of the model objects used with this tool.

3.3.1 Functional Characteristics

These characteristics are not required for determining the electrical behavior of a given device, or for implementing a specific model. Rather, they represent basic functionality which is required of these objects in order to be useful to another object which interacts with them.

Name

Each model should have a descriptive name. In the case of the region-based bulk CMOS model, the name is set to “Bulk Simple Model”.

Ability to Reset Parameters to Defaults

Each model should have the ability to reset all parameters, including fitting parameters, to default values. These default values may be set by the user upon model loading and initialization, and thus might be useful to retain as defaults should the user wish to return to the initial parameter values of the model. In the case of the simple bulk model, this simply resets all of the parameters to their default (possibly user-defined) values.
Ability to Acquire Parameter Properties

Each model parameter has bookkeeping properties associated with it, which include: parameter name, default value, units descriptor, units conversion factor, and default value limits. Internally, all model parameter values are converted to the same set of units, but to the user, the model parameter values should be presented in the most common units.

For the case of the simple bulk model, one typical parameter is the substrate doping. The parameter name is “Na”, the default value is $3.5 \times 10^{23} \text{ m}^{-3}$ ($3.5 \times 10^{17} \text{ cm}^{-3}$), the units descriptor is “cm$^{-3}$”, the conversion factor is $10^6$, and the default value limits are initially set to $10^{21}$ and $10^{25} \text{ m}^{-3}$ ($10^{15}$ to $10^{19} \text{ cm}^{-3}$).

3.3.2 Device Model-related Characteristics

These are related directly to the implementation of a device model and its electrical behavior equations.

Model Equations

Since the model object defines a specific device model, it is obviously necessary to provide some way to acquire the electrical characteristics of the described device. This is done by calculation of drain current, given voltage biases. Model parameters are previously set to the desired values. The device model object must describe the device model equations. For instance, in the case of the bulk simple model, the region-based drain current equations are described, where the specific region of operation is determined by the given voltage biases (e.g. cutoff, triode, or saturation), and the equation corresponding to that region of operation is used to determine the drain current.

Ability to Set Model Parameters Values and Defaults

Each model must provide a procedure for selecting values for all of its parameters, as well as default values for all parameters. Since parameter values are expressed to the user in terms of most common units, but are represented internally by another set of units,
an implicit conversion must be performed (transparent to the user).

**Ability to Acquire Optimization Information**

Optimization which involves a specific device model might converge faster if specific information about the device model were available to the optimizer. For instance, it might be useful to know if a current equation was differentiable with respect to a certain variable. As such, all device model objects are required to have the ability to provide information which could be of potential use to the optimizer. This is done by passing an optimization information object upon request. The optimizer can then extract any information which might be useful for faster convergence. The optimization object protocol is given in the next section.

### 3.4 Optimization Information Object Protocol

An optimization information object describes information which could be of possible use to an optimization routine. It is included in a device model so that any optimizer which uses the electrical behavior equations within the model can be provided with specific information about the equations.

The present system does not incorporate any optimization information into the models, but the framework and protocols have been implemented such that the possibility of specifying such information is available. Incorporation of a dynamic optimizer which recognizes and adapts itself according to properties of model equations is a topic for future research.

Each optimization information object must conform to the protocol described below:

#### 3.4.1 Properties List

One way to describe the particular behavior of a specific function is to provide a list of descriptive properties of that function. Such a list is a set of short descriptors which could be accessed by another object and checked for inclusion of a specific property. For
instance, if a function is strictly concave, it would have only one maximum or minimum. A faster optimization routine which might not work well for functions with multiple extrema might perform well for the case of strictly concave functions. An optimization routine might modify its algorithm to be less computationally expensive before determining the extrema, based on the knowledge that the function is strictly concave.

The descriptor name for a property which does not describe the existence of a function should be in the following format: a name describing the property, an underscore, and the variable name which to property refers to. For instance, if a device model equation was concave with respect to the variable “W”, then the descriptor name would be “concave_W”. If a property is valid with respect to all parameters in a model, then the string following the underscore is “all”. For instance, if a function was monotonic with respect to all of its parameters, the descriptor “monotonic_all” would be used.

Properties which may be of some use to an optimization procedure include monotonicity, concavity, and differentiability.

### 3.4.2 Function Objects List

Certain properties associated with a given function are best expressed as functions. For these properties, the optimization routine should also be able to access such functions. For instance, certain algorithms require the first derivative of a function. If the analytical form of the first derivative is known, it would be computationally less expensive to calculate than a numerical calculation of the first derivative (which nominally would require at least two function evaluations). An optimization routine can retrieve the function and utilize the function within its algorithm.

Functions are represented within optimization objects as function objects. Function objects contain the representation of the function itself, as well as a few bookkeeping properties such as an identifier and a list of variable names. The identifier should be of the form: function name, underscore, variable which the function describes. For instance, the function which describes the first derivative with respect to a parameter “W” would be identified as “firstderivative_W”.
The descriptor name for a property which describes the existence of a function should be in the format: the string “defines_” followed by the function identifier.

To add the first derivative function with respect to the parameter “W” to the optimization object, the descriptor “defines_firstderivative_W” would be added to the properties list, and the function object would be added to the function objects list. The optimizer, upon acquiring the properties list, can search for the descriptor upon finding the desired descriptor, can query the function objects list to acquire the desired function object, and use the function object information to potentially improve the optimization algorithm execution time.

The most useful functions which could be passed to optimization routines are derivative functions.
Chapter 4
Optimization

Optimization is useful in almost any field, and is especially useful in engineering problems. Success is often correlated with the ability to make the best decision quickly and accurately. The use of optimization techniques and algorithms can be used to solve certain problems to achieve the best result.

Optimization deals with two types of problems [ZAHRADNIK]. The first kind of problem is one in which values must be assigned to a set of parameters or independent variables to cause some preassigned function of those parameters to be maximized or minimized. These types of problems are referred to as static optimization problems. If the function to be maximized or minimized depends not merely on the values of a set of parameters but on their trajectories through some sort of space, the objective function in this case is known as a functional, because it depends upon the path of the independent variables. These are often referred to as dynamic optimization problems.

The static optimization problem which occurs within this tool is to determine the optimal set of semiconductor device parameters. The optimization problem can be expressed as the minimization of a specific function of the device parameters, also known as a penalty function. The first step required in solving the problem is the determination of the penalty function. The second step is the determination of the optimization method
which will walk the parameter space to search for the global minimum of the penalty function. The solution found from the global minimum of the penalty function can then be regarded as the “best” solution for the problem.

There are two kinds of optimization which can be performed here: if feasible parameter sets have been generated, then it is possible to simply search through the parameter sets for the specific parameter combination which minimizes the specified penalty function. On the other hand, if it is desired to consider minimization of a weighted penalty function along with a nonstrict electrical specification, then the optimization performed will search the parameter space to find a given minimum of this combined penalty function. In this case, optimization algorithms which are used to search a parameter space (as opposed to simply a set of points) for a minimum are used to find the desired parameter combination. This can be thought of as searching the space “around” a particular parameter curve generated from feasible parameter sets. Figure 10 illustrates the differences between the two methods. The size of the parameter space to be searched around a curve depends on the degree of strictness of the particular electrical specification used to generate the curve.

FIGURE 10. Searching along a curve vs. “around” a curve
4.1 Penalty Function Definition

Often an optimization problem will be expressed in terms of preferences on parameter values. In such cases a specific penalty function which takes into consideration such preferences must be created. The formulation of the correct penalty function is crucial to an accurate solution of a given optimization problem. In the context of this tool, it may be undesirable to have a certain range of values for certain parameters when compared to other parameters (for instance, changing a layout parameter in a circuit design may be more economically feasible than changing a process parameter, which might require using a more expensive process to fabricate a design); thus a penalty function could reflect this by resulting in a large penalty value for parameters which fall into such undesirable ranges. The penalty function which is formulated here allows the user to weight two different characteristics: the desirability to hit a certain performance target, and desirability to express a preference on certain parameter values. In this way it is possible to alter the penalty function based upon certain optimization circumstances.

4.1.1 Least Squares Minimization

A least squares penalty function is used when the object is not to minimize a certain function, but rather to achieve a certain target. For the purposes of semiconductor devices, such a target is a set of known electrical specifications. In this case, the object is not to assign weights to certain parameters, but rather to consider the effects of all parameters as they apply to the evaluation of a given (not necessarily linear) function. In such an instance, the penalty function which is used is the square of the difference between the target value, and the value calculated using the set of parameters to be optimized. Mathematically put, given an equation \( R = f(Q) \) and a target \( R_0 \) (\( Q \) is a set of parameters to be optimized), a least squares type of penalty function would be:

\[
P_F(Q) = (R_0 - f(Q))^2
\]  \hfill (EQ 2)

Note that the minimum of this function is at a set of values \( Q_0 \) where the target \( R_0 \) is achieved. It is important to note that \( Q_0 \) is not necessarily unique, and in fact, for many
functions there are infinitely many solutions for \( Q \) which result in a zero value for the penalty function.

### 4.1.2 Assignment of Weighting Factors

The methodology of defining a specific penalty function depends on the nature of the optimization problem. Another type of penalty function can be constructed which assigns certain multiplicative factors (weights) to each of the different parameters, with this “weighted penalty function” being a simple linear combination of certain parameters. The exact assignment and value selection of the weights would depend on the circumstances of the optimization. Such a function can be expressed as follows:

\[
WPF(P) = \sum \alpha_i \cdot SF(P_i) \tag{EQ 3}
\]

where the summation is over all parameters in the set \( P \), \( P_i \) is the \( i \)th parameter within \( P \), and \( SF \) is a scaling function used to normalize the parameters.

When formulating a weighted penalty function, issues of scaling must be considered. While it is possible to create a desirable weighted penalty function based on weights alone (by calculating the weights such that all parameters are roughly on the same scale), it is often desired to convert parameter values to a comparable scale to avoid the possibility that one parameter might not be considered. This is especially true when typical values of parameters are on very different scales. In the context of semiconductor device parameters, substrate doping levels are usually expressed in values in the order of \( 10^{21} \) m\(^{-3} \), or \( 10^{15} \) cm\(^{-3} \), while most other parameters, such as oxide thickness, are expressed in orders of magnitude less (e.g., oxide thickness is usually on the order of \( 10^{-8} \) m, or \( 10^{-6} \) cm. A weighted penalty function which had equal weights for all parameters would lead to an optimization algorithm which would choose the minimal value for substrate doping since the magnitude of values for that parameter dwarfs the values of all other parameters which can be varied.

One method which may be used to bring all parameters to the same scale would be to use logarithmic (nominally, base 10) values of parameters, instead of the actual parame-
ter values themselves. Such a penalty function would then automatically bring values of parameters down to a more reasonably comparable scale. In the example discussed above, substrate doping values would now be in the range of 21 to 22, while oxide thickness would be in the range of -8 to -9. This scale represents a more appropriate scale than was previously used.

The formulation of a weighted penalty function which fits a particular optimization problem may be difficult depending on the exact nature of the problem and how complicated the “desirability” factor for certain parameter values becomes. For the simpler optimization problems, it is sufficient to specify to minimize or maximize a particular parameter value. For more complicated problems, where it is desirable to penalize specific combinations of parameter values, considerable time may be spent on the exact form and weights within the penalty function.

4.1.3 A Semiconductor Device Parameter Penalty Function

One type of general penalty function which may be applied to the problem of semiconductor device parameter optimization is a combination of both of the aforementioned penalty functions. It can be mathematically expressed as follows:

\[
PF(P) = \sum \left[ \lambda_i (I_{oi} - M(V_{oi},P))^2 \right] + \beta \cdot WPF(P)
\]  

(EQ 4)

where the summations runs over the number of electrical specifications for a given device, \(I = M(V,P)\) is the device model’s function for calculating current, electrical specifications are given as sets of \((I_0, V_0)\), and \(\lambda\)’s and \(\beta\) are the weighting factors for the least squares and weighted penalty functions (not to be confused with the weighting factors within the weighted penalty function). The exact form of the weighted penalty function must be determined based on the circumstances of the optimization. The exact values of the \(\lambda\)’s and \(\beta\) are determined by the nature of the problem, and are used to determine how large of a parameter space around a “feasible parameters” curve is to be searched (See Figure 10).
For instance, if the optimization desired to stress the electrical specifications strongly, while only using the weighted penalty function as a “tie-breaker” between parameter sets which satisfy all electrical specifications, then all values of $\lambda$ would be large compared with values of $\beta$. Also, modification of each separate $\lambda$ value allows the designer to stress certain electrical specifications within the electrical criteria set over others.

This type of penalty function can be generalized even further by allowing for non-linear weighted penalty functions. This can be done by simply replacing the WPF function in Equation 4 with any other desired function. (For instance, another type of penalty function which can be considered is a penalty function which weights parameter deviations from particular parameter values). However, its current form is sufficient enough to implement for the purposes of this tool.

### 4.2 Parameter Constraints

In addition to the formulation of a penalty function, certain parameter constraints can be included in an optimization problem. Such constraints can be in the form of either (1) parameter dependencies which must be enforced and (2) restrictions on parameter ranges.

There may also be constraints on certain parameter values which are either specified in the optimization, or inherent to the nature of the problem itself. In general, there may be two types of parameter value range constraints: ranges of parameter values where functions are valid, and ranges specified when formulating the optimization problem. For the purposes of semiconductor device parameter optimization, these two categories may be described as model validity ranges, and optimization-specific ranges.

The first range (model validity) incorporates the idea that certain parameter values cannot be achieved when applied to real fabrication of semiconductor devices (for instance, substrate doping cannot be higher than $10^{26}$ #/m$^{-3}$, or $10^{20}$#/cm$^{-3}$). There also may be ranges of parameter values over which a certain device model is known to be invalid. Such ranges must be enforced when exploring for optimal device parameters since
values outside of those ranged, while optimal in terms of satisfying the penalty function minimization criteria, are of little practical worth.

The second range represents parameter ranges specified by the formulator of the optimization problem. Such ranges should be more strict than the model validity ranges and should be used to indicate a restriction in the semiconductor device parameter space to be explored. For instance, if the only available fabrication processes having substrate dopings in the range of $10^{16}$ to $10^{17}$ #/cm$^{-3}$, there is little practical use in exploring doping values outside of this range (though it might be useful for research and other theoretical purposes).

4.3 Optimization Methods

There are several optimization methods which have been tested for use with this tool. Each method involves finding the minimum of a multivariable function. Each optimization has its own advantages and disadvantages. Specifically, there is a distinct trade-off between required prior knowledge (about the function to be minimized) and algorithm execution time and efficiency.

4.3.1 Downhill Simplex Method

The downhill simplex method is due to [NELDER]. This method requires only function evaluations, and thus requires no prior knowledge about the function. It is not very efficient in terms of the number of function evaluations that it requires. However, the downhill simplex method may be the best method to use if a quick estimate of a problem is needed [PRESS].

A simplex is a geometrical figure consisting, in N dimensions, of N+1 points and all of their interconnecting line segments. In two dimensions, a simplex is a triangle, and in three dimensions it is a tetrahedron. In general we are only interested in simplexes that enclose a finite non-zero inner N-dimensional volume (all other simplexes are known as degenerate simplexes).
The basic method involves specifying N points (one of the points in the simplex is
assumed to be located at the origin), and iterating while reducing the volume enclosed by
the simplex, until the volume encloses a single point which represents a local minimum.
The volume reduction is achieved by moving the point of the simplex where the function
is largest through the opposite face of the simplex to a lower point. Termination criteria is
determined if the function value as a result of one iteration is less than some fractional tol-
erance of the previous iteration.

4.3.2 Gradient-Based

This method requires knowledge of the gradient of a function. If the gradient inform-
ation is not supplied, this method will calculate a numerical gradient for the required
function at for each iteration (much more computationally expensive than an analytical
form). However, using the gradient information greatly increases the efficiency of the min-
imization algorithm. Using the gradient information, it is possible to determine possible
“directions” towards a minimum in multidimensional space.

The specific minimization method here is called the conjugate gradient method.
For a very simplistic, but inefficient, minimization algorithm, it is possible to travel in
multidimensional space in the direction of the local downhill gradient until a minimum is
reached along that line, and then repeat until the global minimum is found. This method is
known as the steepest descent method. The problem with this method is that new direc-
tions travelled must be perpendicular to old directions, since the new gradient at the mini-
mum point of any line is perpendicular to the direction just traversed. This can lead to
situations where the new direction travelled is less than optimally positioned to locate the
global minimum.

In the conjugate gradient method, the direction of each successive iteration is con-
structed to be conjugate to the old gradient, and attempts to be conjugate to all previous
directions travelled. The specific method implemented is the Polak-Ribiere variation of the
gradient method, in which a sequence of directions is constructed using only line minimi-
izations and gradient information [POLAK]. This method works best for functions that can
be approximated by quadratic forms, but has problems with functions with multiple minima.

4.3.3 Simulated Annealing

The method of simulated annealing is a technique that has attracted significant attention as suitable for optimization problems of large scale, especially where a desired global minimum is hidden among many local minima [KIRKPATRICK]. It thus would work well for penalty functions which incorporate and weight equally both types of penalty functions, which may be more likely to have many local minima, since parameters which minimize one part of the penalty function may not necessarily minimize the other.

At the heart of the method of simulated annealing is an analogy with thermodynamics, specifically the way metals cool and anneal. At high temperatures, the molecules of a liquid move freely with respect to one another. If the liquid is cooled slowly, thermal mobility is lost. The atoms are often able to line up and form a pure crystal that is completely ordered. For slowly cooled system, nature is able to find this ordered state, which is the state of least energy. One interesting point is that, if a metal is cooled quickly, the atoms do not reach this minimum energy state, but rather end up in some polycrystalline state of somewhat higher energy.

The essence of the algorithm is the search for a minimum, while mimicking the slow cooling effect over time. Typically, the slow cooling is modeled as an exponential decrease of some control parameter. Parameters can be adjusted randomly while this control parameter is slowly reduced with an annealing schedule. The parameter set corresponding to the smaller function evaluation is always saved, and the algorithm will occasionally accept a larger function evaluation at high values of the control parameter. Once the control parameter decreases beyond a certain point, the algorithm stops accepting “uphill” steps and only accepts downhill steps.

The effectiveness of this method also depends upon how the random parameter sets are generated. The problem is one of efficiency. A generator of random changes is inefficient if, when a local “downhill” move exists, it always proposes an “uphill” move.
Efficient generators which work well in a general case have not been developed, and thus this method should be used with caution unless a good random parameter generator can be formulated.

For the purposes of this tool, the random parameter generator used is a modification of the downhill simplex method. The “random” new generation of parameters is equivalent to iterative changes in the simplex optimization method.

4.3.4 Exhaustive Search

It is possible to find the minimum of a penalty function by performing an exhaustive search over a restricted set of points in the space. Specifically, if a given constraint is that the electrical criteria for a device must be satisfied, then using this electrical constraint, it is possible to generate a feasible device parameter space via the methods described in Chapter 3, and then exhaustive search the feasible device parameter space to find the desired penalty function minimum. This is the method used if a feasible parameter space has been generated already, since the set of parameter combinations is usually some reasonably small number. It is then possible to completely search the generated set of parameter combinations for a minimum of the penalty function.

If only one electrical constraint in used within the penalty function format given in Equation 4 (and a suitable “weighted” penalty function is given), then the technique of using that electrical constraint to generate a feasible device parameter space and exhaustively searching this space for a minimum of the penalty function may provide the desired global minimum without requiring a large execution time. This is true only if the generated parameter space consists of a reasonably small number of parameter sets.

It should be noted that this technique of exhaustive searching is useful only if the space to be searched has been restricted to be sufficiently small enough to allow for a reasonable execution time. Thus, for most applications where multiple electrical specifications are required, it is best to utilize one of the other optimization method
4.4 Incorporation into System

The specific application of optimization in the system described in this thesis is the use of optimization for the selection of a “best” set of semiconductor device parameters. The general process of selection involves the formulation of a penalty function which describes how certain values for certain parameters are undesirable. Formulation of a specific penalty function can be achieved by creation of a ModelPenaltyFunction object (which conforms to the generic semiconductor device parameter penalty function specified in Equation 4) and specification of all weighting parameters and electrical specifications. Parameter constraints can also be specified, and default to model-validity constraints for each parameter. The optimization method can be specified if a priori knowledge of the behavior of the penalty function is known.

The optimizer module functions as a separate module from the rest of the system, but it is still an integral part of the tool. The idea of semiconductor parameter space exploration and its applications would not be complete without a method of determining not only valid regions of parameter space but also specific points within those regions which conform to a set of optimal characteristics.

The optimization module is implemented as a separate software object. It takes inputs from the main module of the tool, including definitions (i.e. assignment of any weights or electrical specifications) for the penalty functions used, and returns an optimal set of parameters. The user has the ability to restrict optimization over as small a range of parameters as necessary, although there is not complete freedom of choice of parameters to optimize. For instance, fitting parameters are assumed to be fixed for the purposes of semiconductor parameter space exploration, and so such parameters are not included in the set of parameters which are able to be chosen by the user of the tool.

More details of the software implementation of the optimization routines and modules are given in Chapter 5. A test scenarios using the simplex optimization routine is shown in Chapter 6. More information on the optimization routines which were implemented in this tool is given in Appendix A.
Chapter 5
Software Infrastructure

The tool described in this thesis has been implemented in the programming language Java, using Java Development Kit (JDK) 1.02. Java is a relatively new programming language which was developed at Sun Microsystems Inc., and is useful because it is distributed and platform-independent [FLANAGAN]. Thus this tool is able to run on multiple platforms (without requiring recompilation of source code) and is an ideal network application. If set up as a client-server abstraction, it allows the processing to be performed on the client end rather than the server end.

Java is also an object-oriented programming (OOP) language. In an object-oriented system, a class specifies a data structure and methods that operate on the data. Objects are instances of classes. They share the structure and methods with other instances of the same class but have their own instance-specific values of the data. This paradigm is quite different from the procedure-based programming of other non-OOP languages such as C and PASCAL.

Object oriented programming allows for three useful characteristics: inheritance, encapsulation, and polymorphism [BOOCH]. Inheritance is a relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines a hierarchy among classes in which a subclass inherits from one or more superclasses, and augments or redefines the
existing structure and behavior of its superclass. Encapsulation refers to the process of hiding all of the details of an object which do not contribute to its essential characteristics. Normally, the structure of an object is hidden, as well as the implementation of its methods. Polymorphism is a concept in type theory. According to polymorphism, a name, such as a variable declaration, may denote objects of many different classes that are related by some common superclass. Therefore, any object denoted by this name is able to respond to some common set of operations in different ways.

Java classes are organized in sets known as packages. In general, all of the classes within a specific package have characteristics in common. For instance, in the case of the implementation of this tool, all of the model classes, as well as any class which dealt only with models was placed in the package “models”.

### 5.1 Tool Hierarchy

The general package hierarchy for this tool is shown in Figure 11:
The Java language also includes numerous standard classes and packages which can be used as code libraries in different programs (similar to the “include” directive in C). Such classes implement methods which are commonly used by programmers. For instance, the System class provides an interface to system functions such as input, output, and error streams [FLANAGAN].

Java has many useful features which were incorporated into the implementation of this tool. These features include dynamic object loading and interfaces. Dynamic object loading allows new objects to be added to the system within requiring the recompilation of the original system code (compilation of the new object source code is required).

Interfaces are Java’s way of implementing multiple inheritance. They are useful in creating objects which define protocols for other objects to follow. They are also useful for any situation where multiple inheritance might be necessary. The creation of an interface involves declaring, but not defining, methods which other classes must define if those classes implement that interface. Figure 12 shows a sample code listing of a Java interface class and a class that uses that interface.

```java
public interface Shape{
    public int numberSides();
}

public class Rectangle implements Shape{
    public int numberSides(){
        return 4;
    }
}
```

In the example in Figure 12, any class which implements the Shape interface would have to define the method numberSides(). Other methods could take as input a class which implements the Shape interface and call the function numberSides() regardless of the input class’s superclass.
5.2 Implementation Issues

It is anticipated that both the applet and application forms of the tool will be useful. As mentioned before, the applet form can be run from any Web browser and thus allows the tool computation and processing to be distributed to the computer where the browser is running. In its current implementation, the applet does not allow any access to the local file system on which the Web browser is running, for purposes of security. JDK 1.1.x versions have extra security features which do allow some applets to access local files; modification of the system to add such features is a topic for future research.

5.2.1 Applications vs. Applets

Java programs are executed as either applets or applications. Applets are generally executed within a Java-enabled web browser after they are downloaded from a remote site. In general, they do not require a separate window to operate in since the web browser exists as a window for the applet to execute in. On the other hand, applications are invoked from the command line and are handled by the Java interpreter on that specific.
machine. Any application with a graphical user interface must either inherit from the Frame class from the standard Java AWT (Abstract Window Toolkit) package, or instantiate another class which inherits the Frame class [GEARY]

The hierarchical infrastructure of this tool was set up such that the tool could be run from either an application or applet format. As such, it has a graphical user interface which runs from within its own window, and its applet form simply consists of a button for the user to push in order to start the tool and a text screen to display error messages.

The implementation of such an infrastructure for this tool involved the creation of a Java interface, DSETmainShell, to define methods which were common tasks to both an applet and an application, but would be implemented differently for each. For instance, the applet form of this tool includes a text screen within the applet to display information to the user. In application format, this information is displayed on the standard output stream.
Another example is retrieving textual information. An application has fewer security restrictions on it; the user can access files on the local file system. An applet does not allow such access; file information may be downloaded from a URL, however. Thus the common task of retrieving textual information must be implemented differently on the applet and application forms of the tool.

The tool’s main module, DSETmainFrame, is actually implemented as a subclass of the standard JAVA AWT class Frame. DSETmainFrame requires passing a DSETmainShell to it during instantiation. Since an applet must derive from the standard Java AWT class Applet, but an application does not, the use of a Java interface is required in order to implement the multiple inheritance needed to make this infrastructure functional.

![Diagram](image)

**FIGURE 15. Tool Application/Applet hierarchy**

### 5.2.2 Other Implementation Information

The tool was implemented in Java using JDK 1.02. Sun has periodically been providing newer versions of Java for use. The author has decided to keep the implementation in JDK 1.02; during the period of research, the latest versions of common web browsers would not support later versions of Java.

To modify the Java computer code to comply with the new JDK 1.1x standards, it is necessary to rewrite some of the GUI methods, specifically the event handling methods.
according to the new specification. Other methods may have been deprecated (in other words, rendered obsolete), and must be changed to comply with the new specification.

5.3 Important Classes & Packages

The two classes which initiate the tool are DSETApp and DSETApplet. These classes serve only to allow the tool to be run as an application or applet. For more information about these classes, see Section 5.2.1.

There are seven major packages which make up the entire tool. One is the main package, which contains the DSETmainFrame, application, applet, engine, and other “main classes” which make up the tool. The other packages are all extra modules which serve to augment the main module of the tool.

5.3.1 Main Object - DSETmainFrame

The DSETmainFrame object performs the duties of the “main” module, specified in Chapter 2. It is responsible for all of the bookkeeping and other “overhead” duties, including passage of information between other modules. This object also is responsible for initial instantiation of most of the other modules, as well as setup of some of the initial graphical user interface (GUI).

This class inherits from the Frame class to provide an initial window for the formation of the GUI for the tool. It also creates the buttons which can be used to request specific activities of the different system modules (such as performing a run of the exploration engine). It handles all of the user inputs, either directly or indirectly, and ensures that the proper communication of data between modules is performed.

The DSETmainFrame class is responsible for initialization and setup of the system. This includes the instantiation and initialization calls to all of the system module objects, and also bookkeeping of any data generated by the different modules.
The DSETmainFrame implements the InfoPassDialogParent, ParamSelectDialogParent, and PlotPanelParent interfaces. Implementing these interfaces allows the DSETmainFrame class to utilize the graphical user interface classes within the GUI package.

The full constructor for the DSETmainFrame class:

```java
public DSETmainFrame(DSETmainShell p, String title, int width, int height, boolean app)
```

requires the specification of a DSETmainShell-implementing object (both DSETApp and DSETApplet implement this interface), a title for the frame, specifications of the initial width and height of the frame, and a boolean flag denoting whether the tool is being run as an applet or an application. If run as an application, then the features of the tool which require access to the local file system are included when the tool is initialized.

During instantiation the DSETmainFrame object is responsible for:

- instantiation and initialization of the GUI
- instantiation and initialization of the ModelObjectLibraryManager object
- acquisition of the ModelObjects through the ModelObjectLibraryManager
- instantiation and initialization of the DSETEngine object.

During operation of the tool, the DSETmainFrame object is responsible for:

- communication of data between different objects
- processing of user inputs
- data management.

5.3.2 Exploration Engine Object - DSETEngine

The exploration engine class, DSETEngine, implements the parameter calculation and extraction algorithm described in Chapter 2. A device model object must be specified before the algorithm may be called successfully. Also, the electrical criteria for the device
must be selected, and the swept and desired parameter names must be chosen, as well as a
range and step size for the parameter to be swept across. The result is returned in an array
the same length as the array of swept parameter values.

The following important methods are used in the DSETEngine class. The instantiation of this class requires no input parameters. Upon instantiation, the default electrical
criteria settings are drain current of 1 μA for V_d=1 V, V_g=1 V, V_s=0 V, V_b=0 V.

```java
void set_model(DeviceModel dm);
```

The `set_model` method is used to specify the desired model to be used in the
parameter calculation algorithm.

```java
boolean set_sweep_param(String x, double begin, double end, double step);
```

The `set_sweep_param` method is used to select the parameter to be varied. It is
necessary to specify the parameter and define the endpoints of the range and the step size.
The method returns a value of true if the parameter has been set successfully, or false if the
parameter does not exist in the model specified by the method `set_model`.

```java
void set_elec_crit(double i, double voltages[]);
```

The `set_elec_crit` method is used to select the electrical specification for the device
model. It takes as input a value for the output current, and an array of values correspond-
ing to voltages applied to the terminals of the device. The input voltage array values must
be in the specific order: drain, gate, source, and bulk applied voltages.

```java
double [] get_param_space_results(String desired_param_name, double lower_bound, double upper_bound);
```

62
The `get_param_space_results` method performs the parameter extraction algorithm specified in Chapter 2. It calculates an array of values of the desired parameter which correspond to the array of swept parameter values generated using the information specified in the `set_sweep_param` method.

The typical procedure to use this class would be:

1. Instantiate the class
2. Invoke the `set_sweep_param` method to choose a parameter to sweep across, and
   the parameter range and step size.
3. Invoke the `set_elec_crit` method to select electrical criteria.
4. Invoke the `get_param_space_results` method to obtain a calculated parameter array, and
   match with the corresponding points in the swept parameter range to form a two-dimensional feasible parameter space.

   Multidimensional parameter spaces can be formed by querying for multiple parameter range to sweep across and applying this technique recursively. For instance, a three-dimensional parameter space can be obtained by querying for two parameter ranges to sweep across (e.g. oxide thickness from $10^{-6}$ to $20^{-6}$ cm, and channel width from 1 to 10 $\mu$m), and recursively sweeping through both ranges to calculate a third parameter (e.g. substrate doping). In this example, sweeping through the oxide thickness range will generate a two-dimensional parameter space of channel width-substrate doping combinations for each oxide thickness point within the range.

5.3.3 Parameter Representation - GenericParam and ParamSet Objects

5.3.3.1 GenericParam

To represent parameters within models, it is necessary to create a separate object which can be used to represent each specific parameter. These “GenericParam” objects, in their most basic sense, are name-value pairs which can be used to represent a parameter at a given value. There is also extra information which can be specified within the Generic-Param Object which can be useful in certain circumstances.
There are multiple constructors for this class, each of which allow the user to specify different amounts of information within the object. This constructor allows the user to set all possible information relating to a GenericParam object.

```java
public GenericParam(String name, double value, double
    units_conversion_factor, String units_string, double lower_bound,
    double upper_bound)
```

The minimal amount of information which must be specified upon instantiation is the parameter name. All other parameters have default values which are used if not specified upon instantiation.

There are five properties which are encapsulated within a GenericParam object:

1. **Name** - Each parameter must have an identifying name. This information must be included in any instantiation of the GenericParam object. This information can be set and retrieved using the two methods:

   ```java
   void set_name(String name);
   String get_name();
   ```

2. **Current Value** - Each parameter must be set to a specific value. If not specified upon instantiation, its value is set to 0.

   The current value of a parameter can be utilized using the methods:

   ```java
   void set_value(String name, double value);
   void set_value(double value);
   double get_value();
   void set_cvalue(String name, double value);
   void set_cvalue(double value);
   double get_cvalue();
   ```

   The methods which require a String input will only alter the parameter value if the String input matches the name of the GenericParam object.

   A GenericParam object can view a parameter in two different units: most commonly used units, and “global” units. “Global” units refer to the MKS system of units. It is
useful to retrieve parameter values in terms of these global units if different parameters are to be used in the calculation of a formula and standard units are required. On the other hand, when requesting input from a user, the most commonly used units might be the most useful. For instance, substrate doping is usually measured in #/cm$^3$. Thus, when requesting input from a user, it is best to request it in terms of these units. However, when computing formulas, doping is best represented in units of #/m$^3$.

The methods set_cvalue and get_cvalue allow for setting and retrieval of a parameter in terms of its most commonly used units. Such units can be defined by the programmer upon instantiation of the GenericParam object, or can be defined later using the methods set_cf and set_units_str.

3. Default Value - Each parameter must have a default value. The default value for a parameter is set to the current value of the parameter which is set upon instantiation. The following methods are also relevant to the default value:

void set_default_value(double def_value);
void set_default_cvalue(double def_value);
void set_default_value(String name, double def_value);
void set_default_cvalue(String name, double def_value);
void reset_default();

The set_default_cvalue methods allow for setting a default value in terms of the most commonly used units. The method reset_defaults sets the current value of the parameter to the default value. The methods which require a String name parameter as input check that the name is equal to the name of the GenericParam object before altering the default value of that object.

4. Units - Each parameter must have a units conversion factor and a most commonly used units string descriptor associated with it. The default values for these parameters are a conversion factor of 1 and a blank string. The conversion factor is a multiplicative factor used to convert from most commonly used units to global MKS unit. For example, the conversion factor for substrate doping (from #/cm$^3$ to #/m$^3$) is $10^6$. The “most commonly used units” descriptor string would be “#/cm-3".
The following methods are used to set and retrieve units information:

```java
void set_units_str(String x);
String get_units_str();
void set_cf(double conversion_factor);
double get_cf();
```

Units information can also be specified during instantiation of the GenericParam object. If not specified, the default units string is a blank string.

5. **Constraints** - Each GenericParam object also can hold upper and lower bound information about the value of a parameter. These values can be set using the following method:

```java
void set_bounds(double lower_bound, double upper_bound);
```

The upper and lower bound information is used when setting a parameter value. These values are given in most commonly used units. If a parameter value is outside the bounds specified, it is set to the nearest bound.

The method:

```java
double [] get_bounds();
```

can be used to retrieve the bounds information, where the first element in the array is the lower bound, and the second element is the upper bound. There also exists a method:

```java
boolean in_bounds(double value);
```

which returns true if the input value is within the prespecified bounds.

The lower and upper bound values can also be specified during instantiation. If not specified, the default values of 0 and $10^{99}$ are assumed.

5.3.3.2 **ParamSet**

The ParamSet class can be thought of as an encapsulated set of GenericParam objects. This is useful when classification of parameters within another object is desired. For the purposes of this tool, it is desirable to create a distinction between environmental,
process, layout, and fitting parameters; thus, each set of parameters is represented by its own ParamSet object.

The maximal number of GenericParam objects which can be held within a ParamSet object must be specified during instantiation of the ParamSet class. The following important methods are also part of the ParamSet class:

```java
boolean add_param(String name);
boolean add_param(String name, double value);
boolean add_param(String name, double value, double cf, String units);
boolean add_param(String name, double value, double cf, String units,
      double lower_bound, double upper_bound);
```

The `add_param` method is used to add a new GenericParam object to the list. A new GenericParam object is instantiated which has the properties specified in the inputs to this method. If the value, conversion factor, units descriptor string, and range values are not specified, the default values are 0, 1, a blank string, 0, and 1e99, respectively.

```java
void reset_all();
```

The `reset_all` method resets all GenericParam object parameter values to their defaults.

```java
String[] get_param_names();
```

The `get_param_names` method is used to retrieve an array of parameter names of currently within the ParamSet set.

```java
int get_num_params();
int get_max_params();
```

These methods are used to retrieve the current number and maximum number of parameters in the ParamSet object.

```java
boolean param_exists(String name);
```

This method is used to query for the existence of a specific parameter name within a ParamSet object. It returns a value of true if the parameter exists, false if it does not.

```java
GenericParam get_param(String name);
```
The `get_param` method is used to access the actual GenericParam object with the specified parameter name.

All of the GenericParam methods can be called from the ParamSet class by invoking the method with the additional parameter of “String name”. For instance, the method:

```java
void set_cvalue(String name, double value);
```

is equivalent to performing the following line of code:

```java
(get_param(name)).set_cvalue(value);
```

which retrieves the GenericParam object with the specified descriptor from the ParamSet object and sets its value (in most commonly used units) to `value`.

### 5.3.4 Model Objects, the DeviceModel interface

All model equations to be used with this tool must be converted to Java code blocks within a class which implements the DeviceModel interface. Instances of classes which implement the DeviceModel interface will hereafter be referred to as “model objects”. The DeviceModel interface implements the standard protocol procedures and properties specified in the end of Chapter 3. The DeviceModel interface and all model objects which implement this interface are located in the package called “models”.

The DeviceModel interface declares the following public methods, each of which must be defined in any model object. Note that there are two methods which can be used to alter a parameter value, in terms of its most common units. and in terms of a global standard scale.

**DeviceModel methods (Complete list):**

```java
double calc_i(double volts[]);
```

This method is responsible for the calculation of the current of the device. The names `calc_i` and `volts` given to the method and its variables are arbitrary, and can be renamed to allow for generic use of the DeviceModel interface to other devices. For the purposes of the four-terminal semiconductor device, the voltages in the array must be in
the order: drain, gate, source, and bulk applied voltages, all given with respect to some global ground plane. The return value of the method is the drain current which occurs as a result of the specified voltages on the terminals of the device.

String get_name();

This method returns a String whose value is the given name of the model object.

void set_param_value(String name, double value);
double get_param_value(String name);
void set_param_cvalue(String name, double val);
double get_param_cvalue(String name);

The method set_param_value is used whenever a parameter’s value, in terms of internal scale of units (i.e. MKS units), is to be changed. To change a parameter’s value in terms of its most common units, use the method set_param_cvalue. The method get_param_value returns the value of a parameter, in terms of internal units. The set_param_cvalue and get_param_cvalue methods are used to set and retrieve a specific parameter’s value, in terms of the parameter’s most common units.

double get_sec_param(String name);

The get_sec_param method returns the value of a specified secondary parameter. For instance, the body factor “gamma” is a parameter which is calculated from the values of substrate doping and oxide thickness. Once the value of those primary parameters have been set, the value of “gamma” may be calculated. Secondary parameters usually represent intermediate quantities which are computed as a part of the process involved in the calculation of the drain current.

double get_cf(String name)
String get_unit_str(String name);

The get_cf and get_unit_str methods are used to retrieve the conversion factor and units string descriptors for the “most commonly used units” for the parameter with the given name. This is useful information which can be presented to the user of the tool, allowing the user to input parameter values in terms of the most commonly used units.
GenericParam get_param(String name);

The `get_param` method is used to access the actual GenericParam object with the specified parameter name.

boolean param_exists(String name);

The `param_exists` method is used to query if a parameter with a specific name is included in the set of parameters for a given model.

String [] get_param_names(String descriptor);

The `get_param_names` method is used to retrieve an array containing names of parameters within a given parameter type set (e.g., environmental, process, layout, or fitting). If called with an input of “all”, it should return an array containing names of parameters in all sets.

void set_default_value(String name, double val);

The `set_default_value` method is used to set a parameter’s default value, in terms of its most commonly used units.

void set_defaults();

The `set_defaults` method is used to reset all parameter values to their default settings. This can be achieved by invoking the `reset_defaults` method in each of the ParamSet objects within the model object.

void set_bounds(String name, double lower_bound, double upper_bound);
double [] get_bounds(String name);
The method `set_bounds` is used to provide a constraint on the range of values a specific parameter can take. If a parameter is requested to be set outside the bounds, then the value of the parameter is set to the nearest bound. This method calls `set_bounds` within the GenericParam object which has the specified descriptor. The method `get_bounds` retrieves the bounding information for a specific parameter. The lower bound is the first element in the returned array, and the upper bound is the second element in the returned array.

```java
void update_all();
```

The `update_all` method is used to update all variables within a model, including secondary variables which may be calculated from the parameters within the model’s parameter sets.

```java
void print_all_params();
```

The `print_all_params` method prints all parameters and values to standard output.

```java
public DeviceModel make_copy();
```

When a model object invokes its `make_copy` method, a copy of that specific model object is returned.

```java
public OptimizationInfo optimization_info();
```

The `optimization_info` method returns an optimization information object. This object describes extra information about the model equations, if such information is known. In this way an optimization routine may query the model object for information about its equations, and can attempt to use such information to expedite the optimization procedure.
5.3.5 The Model Object Library Manager - How to Add Models

The Model Object Library Manager is a class which is responsible for initialization and original instantiation of all model objects. It utilizes dynamic class loading to create an array of DeviceModel-implemented objects. The actual model objects are not hard-coded into the system, but rather are determined only from a model information text file which is read by the ModelObjectLibraryManager. From the information within the file, the ModelObjectLibraryManager attempts to create instances of each model object specified in the file. The ModelObjectLibraryManager is part of the package "models".

The most important method within the ModelObjectLibraryManager class is the method:

```java
void load_models(DataInputStream din);
```

This method takes as input an object of type DataInputStream, which is a standard JAVA class which represents a stream of data extracted from either a file (for applications) or a URL (for applets). The exact input is determined when the tool is executed - the DSETmainShell interface declares a method which must supply the required DataInputStream object, and the DSETApp and DSETApplet classes create such a DataInputStream.

The input for the method can be loaded from an ASCII text file. Such a file follows a specific format which is used to specify the names of the model objects to be used in the tool, as well as the default parameters for each model object. The format is given in Figure 16.

The procedure for adding a new model object to the list of objects to be used with this tool is as follows:

1. Convert the model equations to Java code, insuring that the Java code follows the model object protocol specified in the DeviceModel interface.
(2) Compile the Java code into a model object.

(3) Move the model object into the models package.

(4) Add the appropriate line(s) to the model file.

Once the procedure has been followed, execution of the tool should lead to incorporation of the new model object. Note that recompilation of the system source code is NOT required in this procedure.

Once the procedure has been followed, execution of the tool should lead to incorporation of the new model object. Note that recompilation of the system source code is NOT required in this procedure.

<table>
<thead>
<tr>
<th>DSET MODEL CLASS INFORMATION</th>
<th>DSET MODEL CLASS INFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;MODEL CLASS NAME 1&gt;</td>
<td>BulkSurfPotModel</td>
</tr>
<tr>
<td>DEFAULTS</td>
<td>DEFAULTS</td>
</tr>
<tr>
<td>&lt;PARAM NAME 1&gt;</td>
<td>&lt;PARAM NAME 1&gt;</td>
</tr>
<tr>
<td>&lt;VALUE&gt;</td>
<td>&lt;VALUE&gt;</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
</tr>
<tr>
<td>&lt;MODEL CLASS NAME 2&gt;</td>
<td>SOIASModel</td>
</tr>
<tr>
<td>DEFAULTS</td>
<td>DEFAULTS</td>
</tr>
<tr>
<td>&lt;PARAM NAME 1&gt;</td>
<td>tox</td>
</tr>
<tr>
<td>&lt;VALUE&gt;</td>
<td>1e-7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
</tr>
</tbody>
</table>

Notes:

1. Specification of default values of parameter is optional
2. Delimiters between parameter name and value can be any form of whitespace (spaces or tabs).

FIGURE 16. Sample model file template

5.3.6 Optimizer Module & Relevant Classes

5.3.6.1 Optimizer class

The Optimizer class adapts the optimization techniques specified in Chapter 4 to computer code. All optimization routines were ported to the Java language from previ-
ously written routines from Numerical Recipes in C. [PRESS]. The Optimizer class and other classes discussed here are part of the package “optimizer”, unless otherwise specified.

All of the optimization routines are written as static methods. This allows invocation of the method through the class itself rather than an instantiation of the class. For instance, the static method simplex() within the Optimizer class is properly invoked as “Optimizer.simplex().”

The following static methods are defined in the Optimizer class:

```java
double [][] simplex(double p[][], double ftol, MultivariableFunction mvf);
```

The `simplex` method implements the downhill simplex method described in Section 4.3.1. The inputs to this method are the initial simplex `p` in N dimensions, given as a two-dimensional array of size (N+1) by N, a value `ftol` which is used in the termination criteria for the algorithm, and a MultivariableFunction-implementing object which represents the function to be minimized. The function returns the final simplex; the first point in this simplex represents the minimum of the specified multivariable function.

```java
double [] gradmin(double p[,double ftol, Multivariablefunction mvf, GradientFunction gf);
```

The `gradmin` method implements the gradient-based minimization method specified in Section 4.3.2. The inputs to this method are the initial starting point `p`, a value `ftol` which is used in the termination criteria for the algorithm, a MultivariableFunction-implementing object which represents the function to be minimized, and a GradientFunction-implementing object which represents the gradient of the function to be minimized. If the GradientFunction argument is not included when this method is invoked, a gradient function is calculation based on the multivariable function passed to the method.

```java
double [][] anneal(double p[][], double pb[],double yb[],double ftol, MultivariableFunction mvf,double temptr,int iters)
```

74
The *anneal* method performs the simulated annealing algorithm described in Section 4.3.3. This method takes several inputs. The first input is an initial simplex $p$, which is given as an $(N+1)$ by $N$ array of values. The inputs $pb$ and $yb$ are arrays which represent the best function point and value found by the algorithm ($pb$ is expressed as an array to simulate passing the variable by reference, which requires this modification due to the lack of use of pointers in Java). The input $ftol$ is the fractional convergence tolerance to be achieved in the function for an early return. The input $mvf$ is a MultivariableFunction-implementing object (e.g., a ModelPenaltyFunction object) which describes the penalty function to be minimized. The algorithm makes $iters$ function evaluations at an annealing temperature $temptr$, then returns an $(N+1)$ by $N$ matrix representing the final simplex.

```java
double [] exhaust_search(double space[][], MultivariableFunction mvf);
```

The *exhaust_search* method implements an exhaustive search for a minimum. The input $N \times M$ matrix $space$ represents a parameter space of $M$ parameter sets, and $mvf$ represents the penalty function of $N$ parameters. The method returns the parameter set which corresponds to the smallest penalty function evaluation within the input parameter space.

### 5.3.6.2 MultivariableFunction interface

The MultivariableFunction interface specifies a protocol for defining multivariable functions. This interface is part of the package “Functions” and can be used to describe any function of more than one variable. A ModelPenaltyFunction object implements the MultivariableFunction interface.

All classes which implement the MultivariableFunction interface must define the following public methods:

```java
double evaluate(double x[]);
```

The *evaluate* method is used to describe the multivariable function. The array of input values $x$ represents the different input parameters for the function.

```java
void set_variables(String var_names[]);
```

75
The set_variables method is used to specify names for each of the values which are passed to the evaluate method. The corresponding name-value pair can be used to set that particular parameter value in a device model using the set_param_value method.

```java
public String get_name();
public void set_name(String name);
```

These two methods are used to set and retrieve a descriptive name for the function. This may be useful in cases where a descriptor for a function is desired (e.g., querying a user for selection of a specific function).

### 5.3.6.3 ModelPenaltyFunction class

The ModelPenaltyFunction class implements the MultivariableFunction interface, and also has additional methods to allow for proper specification of a penalty function. The evaluate() method defined in this class implements the device parameter penalty function shown in Equation 4 of Section 4.1.3.

The following important methods are defined in this class:

```java
public ModelPenaltyFunction(int max_elec_specs, int number_variables);
```

The constructor for the ModelPenaltyFunction class requires the specification of the maximum number of electrical specifications and the number of parameters which can be varied.

```java
void add_elec_spec(double i, double voltages[]);
```

The add_elec_spec method is used to add an electrical specification to the penalty function. The voltage values within the 4-element array voltages must be in the order: drain, gate, source and bulk, respectively.

```java
void set_weights(double weights[]);
```

The set_weights method is used to select the weights for the parameters which are to be optimized. These weights are the \( \alpha \)'s in Equation 3 in Section 4.1.2.
5.3.6.4 GradientFunction interface

The GradientFunction interface is used to describe a gradient function. The gradient for an N-dimensional function takes as input an N-dimensional array corresponding to a single point in space, and computes in an N-dimensional gradient vector. All functions which implement the GradientFunction interface must define the function:

```java
double [] evaluate(double x[]);
```

The evaluate method should contain the equations for analytical calculation of the gradient of a specific function. If an analytical form is not known, then it is possible to use the class CalcGradientFunction, which will calculate a numerical gradient for a given function.

The GradientFunction also declares the same methods as in the MultivariableFunction interface. The only modification is to allow the evaluate method to return a point in multidimensional space rather than a single value.

5.3.6.5 CalcGradientFunction class

The CalcGradientFunction class implements the GradientFunction interface and can be used to provide a GradientFunction-implemented object to the gradient-based minimization method `gradmin` if an analytical form of the gradient of a function is not known. The multivariable function whose gradient is to be numerically calculated must be passed to this object upon instantiation using the constructor:

```java
public CalcGradientFunction(MultivariableFunction mvf);
```

When the evaluate method of this class is invoked, the gradient is calculated numerically by the usual gradient formula:

\[ V_i F(p) = \frac{(F(p + \Delta p, R) - F(p, R))}{\Delta p} \]  

(EQ 5)

where \( i \) is one of the axes of dimensions (e.g. x, y, or z axis), \( p \) is the value of the evaluation point along the \( i^{th} \) axis, \( \Delta p \) is a small step size along the \( i^{th} \) axis, \( R \) represents the set
of values of the evaluation point not on the $i^{th}$ axis, and $F$ is the multivariable function whose gradient is desired. The formula is used $N$ times for an $N$-dimensional function, to find the gradient in each of the $N$ dimensions. The resulting array of values is returned by the `evaluate` method.

### 5.3.6.6 OptimizationInfo class

The OptimizationInfo class provides potentially useful information which can be supplied to an optimizer. It implements the optimization information object protocol specified in Section 3.4.

The following important methods are defined in the OptimizationInfo class:

```java
void add_property(String descriptor);
void add_property(String descriptor, MultivariableFunction mvf);
```

The `add_property` method is used to add descriptors and function objects to the list of optimization information. The list of descriptors and the list of function objects are kept in separate data structures and can be retrieved using methods described in this section.

If a function property descriptor (such as "monotonic_tox") does not require a function object, then the `add_property` method is invoked with simply a descriptor. Other properties may require specifying a MultivariableFunction object in addition to the descriptor.

For example, adding a gradient function for a specific parameter (e.g., tox, the oxide thickness) would require invoking this method with a descriptor of "defines_gradient_tox", and a MultivariableFunction object corresponding to the first derivative of the I-V function with respect to oxide thickness. This MultivariableFunction object should have a suitable descriptor, set using the `set_name` method of the MultivariableFunction interface protocol.

```java
String []get_properties();
```

The `get_properties` method is used to retrieve the list of descriptors for a given model. This list can then be searched for desirable property descriptions.
MultivariableFunction get_function(String name);

The *get_function* method is used to retrieve a MultivariableFunction object with a specific descriptor. Each function which is in the functions list of the optimization information object can be queried for its name, which can be compared against the desired function name. A matching function name would result in that MultivariableFunction object being returned. If a matching function name is not found, then the method returns a null object.

### 5.3.7 DSET Graphical User Interface (GUI) Package

The graphical user interface package contains classes which are used to handle user input. Most of the user input is either directly or indirectly handled by the DSET-mainFrame class (the exception being the plotting utility inputs, which are handled by the plotting classes) through information which is gathered by graphical user input modules from this class.

The graphical user input modules elements allow the user to specify information which is used in the process of determining the feasible parameter space. These modules are simply dialog boxes which allow information to be entered into text fields to be passed to the DSET-mainFrame object. These graphical elements are spawned when the tool user selects an option which requires the input of such information.

The other classes of interest within the graphical user interface package are the PlotCanvas and PlotPanel classes. The Plot classes form the basis of the output display module.

#### 5.3.7.1 Plotting Utilities (PlotCanvas and PlotPanel)

The PlotCanvas class is a plot output display object which plots standard X-Y line graphs. It inherits from the standard Java AWT class Canvas, which is used for most line and shape drawing displays. This object consists solely of the line graph and its accompanying axes; all other functionality which is included must be invoked via method calls.
The PlotCanvas class uses buffered imaging. This means that once the image is drawn it is saved to an Image object buffer. When the window containing the PlotCanvas object is altered (moved, redisplayed, etc.), the buffered Image object is redrawn to the screen. This avoids certain image refresh problems which may occur if an unbuffered image is used.

The PlotCanvas class includes methods which access the following features:

- Multiple graph data storage
- Multiple data sets per graph
- Autoscale capability
- Color selection

The PlotPanel class serves as a “wrapper” around the PlotCanvas class, while adding the ability to access additional functionality. Since the standard Java Canvas class (which is a superclass of the PlotCanvas class) does not allow the addition of graphical widgets such as buttons, it is necessary to create another class which allows for graphical widget access to all of the methods within the PlotCanvas class. The PlotPanel class inherits from the standard Java AWT class Panel, and thus can be embedded into any other Panel or Frame class (such as DSETmainFrame). It implements the PlotCanvasParent interface, and consists of the PlotCanvas object and additional buttons which invoke all of the graphical feature methods within the PlotCanvas object.

The constructors for the PlotCanvas and PlotPanel classes are as follows:

```java
PlotCanvas(PlotPanelParent f, int numgraphs, int numcurves, int width, int height);
PlotPanel(PlotPanelParent f, int numgraphs, int numcurves);
```

The PlotCanvas constructor requires the specification of an object which implements the PlotCanvasParent interface. The PlotPanel constructor requires the specification of an object which implements the PlotPanelParent interface. All other parameters within the constructors are optional. The parameter `numgraphs` can be used to specify the maximum number of graphs which can be stored. Its default value is 10. The parameter `numcurves` specifies the maximum number of curves which can be included on a single graph.
Its default value is 20. The parameters width and height describe the initial height and width of the plot, in units of pixels. The default values are 560 and 240, respectively.

The PlotPanelParent and PlotCanvasParent interfaces do not currently declare any methods. They were implemented to allow for simpler potential future developments.

The following is a list of methods which are defined in the PlotCanvas class and invoked through the PlotPanel class. These methods implement the graphical features described earlier in this section.

```java
void set_current_graph(int i);
```

The `set_current_graph` method is used to select the current graph to be displayed. The PlotPanel object presents a button to the user which, when selected, spawns a dialog box which allows the user to select a particular graph. The PlotPanel object then invokes this method.

```java
void clear_data(int i);
```

The `clear_data` method is used to erase all data currently held in a specified graph. If the integer graph index is not given, then the data on the graph currently displayed is erased. The option to clear the data is presented to the user by a button in the PlotPanel object, and invoked by the PlotPanel object when the user selects the button.

```java
void set_xaxis_limits(double xmin, double xmax);
void set_yaxis_limits(double ymin, double ymax);
```

The methods `set_xaxis_limits` and `set_yaxis_limits` are used to set the range of the plot’s axes. The option to set the axis limits is presented to the user by a button which, when selected, spawns a dialog box which allows the user to specify values for the axis limits. The PlotPanel object then invokes the appropriate methods with the given inputs.

```java
void autoscale();
```

This method is used to adjust the plot’s axes such that all curves can be fully seen. The option to autoscale the plot is presented to the user by a button in the PlotPanel object, and invoked by the PlotPanel object when the user selects the button.
void set_current_color(Color c);

The set_current_color method is used to select a graphing color which would be used to draw the next set of data points to be plotted. The option to select a color is presented to the user by a menu in the PlotPanel object, and invoked by the PlotPanel object when the user selects the menu and scrolls through the color choices.

Other important methods in the PlotCanvas class are the methods:

set_data(int whichgraph, double x[], double d[], double val, int whichline, Color c);
add_data(int whichgraph, double x[], double d[], double val, Color c);

The set_data and add_data methods are used to specify data to be plotted. The method add_data invokes set_data such that the data is stored in the next available curve slot on the graph. Invoking these methods requires the specification of two arrays of values \( x \) and \( d \). The \( x \) values represent values along the \( x \) axis, and the \( d \) values represent corresponding values along the \( y \) axis. The whichgraph parameter is a graph index which indicates the graph to add the data to (default: current graph number). The val parameter is a parameter used to indicate an identifying value for a particular curve in a graph (default 0). The whichline parameter specifies the curve index on the graph to store the data. It must be specified when the set_data method is invoked. It is also possible to specify a color to plot the curve on the graph. If not specified, the color default is the current color (which can be set using the set_current_color method).
Chapter 6

Test Scenarios

As outlined in Chapter 1, there are various problems and areas which can be explored using this system. Several scenarios have been formulated to explore the possibilities available to the user of this tool. It should be noted that the accuracy of the device models used in the regions of interest are directly correlated with the accuracy of results obtained with this tool. Again, there is the trade-off of time vs. accuracy: for the case studies presented here, a less accurate model can provide an initial estimate of the solution, but for improved accuracy, an appropriate device model should be used.

6.1 Case 1: Selection of Device Structure

Problem Statement:

Given a set of fixed electrical specifications for a circuit, what are some feasible device parameters within a given technology which can be used in this circuit design? Which processes can be used to fabricate devices which meet these specifications?

Problem Statement:

Given a fixed electrical specification, what are the parameter dependencies which exist in a specified device for a specific technology? For instance, assuming fixed electrical...
specifications, what change in oxide thickness would compensate for a given change in substrate doping?

**Problem Statement:**

*Given a fixed electrical specification, what is the best technology to use given a particular range of process parameters? In other words, how do different technologies compare to each other given the constraint of a fixed electrical specification?*

These problems are basic questions which can be answered by feasible parameter space generation. The electrical criteria for each transistor can be used to generate a series of feasible parameter combinations.

For example, assume that the layout parameters for a particular device have already been set (L=0.5 μm, W= 1 μm), and it is desirable to explore what feasible parameter combinations are for this device. For a given electrical specification, it is possible to generate feasible parameter combinations for a given range of parameters. Suppose that the only processes available all have substrate dopings between $10^{16}$ cm$^{-3}$ and $10^{17}$ cm$^{-3}$, and it is desirable to explore parameter combinations of substrate doping and oxide thickness. Using the electrical criteria for a given device with the exploration tool will generate the feasible parameter combinations within the specified range. The generated data is shown in Table 1. The data was generated for drain current of 200 μA for $V_d = 1$ V, $V_g = 1$ V, $V_s = 0$ V, and $V_b = 0$ (V$_{ds} = 1$ V, V$_{gs} = 1$ V, and V$_{bs} = 0$ V) for the charge-sheet model described by Equation Set 2 of Section 3.1.1.

**TABLE 1. Feasible Parameter Combinations for a bulk CMOS device**

<table>
<thead>
<tr>
<th>$N_a$ x $10^{16}$ cm$^{-3}$</th>
<th>$t_{ox}$ (nm)</th>
<th>$N_d$ x $10^{16}$ cm$^{-3}$</th>
<th>$t_{ox}$ (nm)</th>
<th>$N_a$ x $10^{16}$ cm$^{-3}$</th>
<th>$t_{ox}$ (nm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>8.4</td>
<td>4.2</td>
<td>6.9</td>
<td>7.4</td>
<td>6.2</td>
</tr>
<tr>
<td>1.4</td>
<td>8.1</td>
<td>4.6</td>
<td>6.8</td>
<td>7.8</td>
<td>6.2</td>
</tr>
<tr>
<td>1.8</td>
<td>7.9</td>
<td>5.0</td>
<td>6.7</td>
<td>8.2</td>
<td>6.1</td>
</tr>
<tr>
<td>2.2</td>
<td>7.7</td>
<td>5.4</td>
<td>6.6</td>
<td>8.6</td>
<td>6.0</td>
</tr>
<tr>
<td>2.6</td>
<td>7.5</td>
<td>5.8</td>
<td>6.5</td>
<td>9.0</td>
<td>6.0</td>
</tr>
<tr>
<td>3.0</td>
<td>7.3</td>
<td>6.2</td>
<td>6.5</td>
<td>9.4</td>
<td>5.9</td>
</tr>
<tr>
<td>3.4</td>
<td>7.2</td>
<td>6.6</td>
<td>6.4</td>
<td>9.8</td>
<td>5.9</td>
</tr>
<tr>
<td>3.8</td>
<td>7.1</td>
<td>7.0</td>
<td>6.3</td>
<td>10.2</td>
<td>5.8</td>
</tr>
</tbody>
</table>
It is possible to repeat the parameter space generation for a different technology in order to analyze and compare possible parameter combinations.

A simple analysis of the information in Table 1 leads to certain information about the parameter dependence of oxide thickness and substrate doping, if the given electrical specification and other device parameter values are to remain the same. For instance, for a substrate doping of $5 \times 10^{16}$ cm$^{-3}$, reducing the doping to $3 \times 10^{16}$ cm$^{-3}$ would require increasing the oxide thickness from 67 to 73 Å, if the same electrical specifications for the transistors are to be met (and the parameter values for other parameters are to remain constant).

Another test scenario was performed where oxide thickness and channel width were explored, while keeping other parameters at the same values, for bulk CMOS and SOIAS models. The electrical biases here were $V_d = 1$ V, $V_g = 1$ V, $V_s = 0$ V, $V_b = 0$ V for $I_d = 100$ μA. Common parameters between the two models, such as substrate doping (set to $3.5 \times 10^{17}$ cm$^{-3}$) and channel length (set to 1 μm) were set to the same values. The resulting parameter sets are shown in Table 2. A plot of the generated data, plotted in MATLAB, is shown in Figure 17.

Comparison of the two device parameter spaces shows that for values of $t_{ox}$ between 5 and 9 nm, SOIAS devices require a smaller channel width to achieve the same

<table>
<thead>
<tr>
<th>$t_{ox}$(nm)</th>
<th>bulk W(μm)</th>
<th>SOIAS W(μm)</th>
<th>$t_{ox}$(nm)</th>
<th>bulk W(μm)</th>
<th>SOIAS W(μm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>1.3</td>
<td>1.2</td>
<td>7.2</td>
<td>3.9</td>
<td>2.4</td>
</tr>
<tr>
<td>5.2</td>
<td>1.5</td>
<td>1.2</td>
<td>7.4</td>
<td>4.3</td>
<td>2.6</td>
</tr>
<tr>
<td>5.4</td>
<td>1.6</td>
<td>1.3</td>
<td>7.6</td>
<td>4.8</td>
<td>2.8</td>
</tr>
<tr>
<td>5.6</td>
<td>1.8</td>
<td>1.4</td>
<td>7.8</td>
<td>5.3</td>
<td>3.0</td>
</tr>
<tr>
<td>5.8</td>
<td>1.9</td>
<td>1.5</td>
<td>8.0</td>
<td>5.9</td>
<td>3.3</td>
</tr>
<tr>
<td>6.0</td>
<td>2.1</td>
<td>1.6</td>
<td>8.2</td>
<td>6.6</td>
<td>3.5</td>
</tr>
<tr>
<td>6.2</td>
<td>2.4</td>
<td>1.7</td>
<td>8.4</td>
<td>7.4</td>
<td>3.8</td>
</tr>
<tr>
<td>6.4</td>
<td>2.6</td>
<td>1.8</td>
<td>8.6</td>
<td>8.4</td>
<td>4.1</td>
</tr>
<tr>
<td>6.6</td>
<td>2.9</td>
<td>2.0</td>
<td>8.8</td>
<td>9.5</td>
<td>4.5</td>
</tr>
<tr>
<td>6.8</td>
<td>3.2</td>
<td>2.1</td>
<td>9.0</td>
<td>10.8</td>
<td>4.9</td>
</tr>
<tr>
<td>7.0</td>
<td>3.5</td>
<td>2.3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
electrical characteristics. This information may be used to make a decision about the particular technology to use to fabricate a particular design. This analysis may be repeated for each device in a circuit, and the results compiled to observe if a technology offers an advantage over another for the purposes of fabrication of a certain design (e.g., if one technology requires consistently lower channel widths than another for the same electrical specifications, it may be advantageous to fabricate using that technology since there would be lower overall capacitance within the circuit). The information provided by the exploration tool can be combined with other information known about the problem (for instance, cost of using a particular technology) to make a decision for particular parameter combinations to use within the circuit.

![Channel Width-Oxide Thickness combinations](image)

**FIGURE 17. MATLAB plot of Channel Width-Oxide Thickness combinations**
6.2 Case 2: Sensitivity Analysis

Problem Statement:

Given a semiconductor device with specific parameter values, how does the output drain current vary with small variations in process parameters?

The problem of analyzing a particular device’s sensitivity to a given parameter can be of some use during early circuit design stages (e.g., providing information on the amount of guard-banding required to allow for estimated process variations). The compact models used with this tool can provide a quick estimate for the amount of current variation which occurs given a particular process parameter variation.

For instance, if a given oxide thickness (e.g. 10 nm) was known to vary in past fabrication runs by a total of 4% (or ±2%), it is possible to use this information with the tool to determine the current variation for a given transistor. Table 1 shows generated data of output current for variations of oxide thickness. The data was generated for drain and gate voltages of 1 V and source and bulk voltages of 0 V (i.e. $V_{gs}=1$ V, $V_{ds}=1$ V, $V_{bs}=0$ V). The model used was the charge-sheet model described by Equation Set 2 of Section 3.1.1.

The fixed parameters used were: a substrate doping of $1 \times 10^{15}$ cm$^{-3}$, a front gate doping of $1 \times 10^{19}$ cm$^{-3}$, a junction depth: $5 \times 10^{-7}$ m, a channel width of 2 μm, and a channel length of 1 μm.

<table>
<thead>
<tr>
<th>$t_{ox}$(nm)</th>
<th>$I_d$(μA)</th>
<th>$t_{ox}$(nm)</th>
<th>$I_d$(μA)</th>
<th>$t_{ox}$(nm)</th>
<th>$I_d$(μA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.80</td>
<td>10.55</td>
<td>9.94</td>
<td>9.47</td>
<td>10.08</td>
<td>8.47</td>
</tr>
<tr>
<td>9.82</td>
<td>10.39</td>
<td>9.96</td>
<td>9.32</td>
<td>10.1</td>
<td>8.34</td>
</tr>
<tr>
<td>9.84</td>
<td>10.23</td>
<td>9.98</td>
<td>9.18</td>
<td>10.12</td>
<td>8.20</td>
</tr>
<tr>
<td>9.86</td>
<td>10.07</td>
<td>10.00</td>
<td>9.03</td>
<td>10.14</td>
<td>8.07</td>
</tr>
<tr>
<td>9.88</td>
<td>9.92</td>
<td>10.02</td>
<td>8.89</td>
<td>10.16</td>
<td>7.94</td>
</tr>
<tr>
<td>9.9</td>
<td>9.77</td>
<td>10.04</td>
<td>8.75</td>
<td>10.18</td>
<td>7.81</td>
</tr>
<tr>
<td>9.92</td>
<td>9.62</td>
<td>10.06</td>
<td>8.61</td>
<td>10.2</td>
<td>7.68</td>
</tr>
</tbody>
</table>

The data above shows the typical inverse relationship of drain current to oxide thickness. For this example it can be shown that for oxide thickness variations of ±2% (or
± 2 Å) for the parameter values and conditions given, a maximum variation of 17% (or 1.52 μA) occurs in the device drain current. This analysis can be repeated using different voltage conditions and parameter values to find out variations for other circumstances.

Multidimensional parameter analysis of current variation is not included within the scope of this tool. Such analysis would involve computation and analysis of parameter-current spaces (versus the pure parameter spaces analyzed here) and is a topic for future research.

6.3 Case 3: Curve Fitting & Parameter Extraction

Problem Statement:

*Given a set of fixed electrical specifications, a set of fixed parameter values, and a specified technology (i.e. specific device model), what is the best set of parameters which meets the specifications as closely as possible?*

This problem is similar to having the penalty function as specified in Equation 4 of Section 4.1.3, with β equal to 0. A typical approach to solving this problem requires the use of a time-consuming numerical parameter extractor. However, using parameter space exploration techniques, it is possible to find a reasonably good guess for parameter values.

As a test case, I-V data was generated from the charge-sheet model described in Section 3.1.1, for a fixed set of layout and process parameters (W=2 μm, L=1 μm, N_a=7x10^15 cm^-3, t_ox=8x10^-7 cm). The following three data points were taken from the data set:

<table>
<thead>
<tr>
<th>V_d</th>
<th>V_g</th>
<th>V_s</th>
<th>V_b</th>
<th>I_d(μA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>0.5</td>
<td>0</td>
<td>533</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0.5</td>
<td>0</td>
<td>331</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
<td>0.5</td>
<td>0</td>
<td>256</td>
</tr>
</tbody>
</table>

Using the tool, it is possible to generate parameter spaces using these electrical criteria, which results in Figure 18. Each curve shown in the figure represents a set of sub-
strate doping-oxide thickness parameter combinations for each of the electrical specifications given in Table 4. The area near intersection of the parameter space curves on the graph denotes parameter combinations which satisfy all three electrical criteria. Notice that this area includes the parameter combination which was used to generate the data, namely, \( N_a = 7 \times 10^{15} \text{ cm}^{-3} \) and \( t_{ox} = 8 \times 10^{-7} \text{ cm} \). There is some inaccuracy due to round-off error which occurs when inputting electrical criteria into the tool, but by visual inspection it is possible to select a good estimate for the parameter values.

Computationally, it may be best to feed the parameter space information into an algorithm to calculate the point in terms of a least-squares approach, finding the point which has the least-squares error of distance from each curve. The implementation of this method is left as a suggestion for future work.

**FIGURE 18. Parameter Space Exploration for Parameter Extraction**
6.4 Case 4: Present and Future Technology Exploration

Problem Statement:

Given a set of fixed electrical specifications, and a device model for a future technology, what are feasible device structures (if any) within the technology which meet the specifications?

Problem Statement:

Given a set of desired electrical specifications (known by a circuit designer), and a specified penalty function of device parameters, what are the optimal sets of parameters for a particular technology family?

Exploring the feasibility of future technologies can be accomplished by a series of feasible parameter set generations using this tool. The silicon-germanium device model implemented is an example of a model which describes a device which is not commonly manufactured yet holds great potential as an option for future VLSI designs. Assuming a given circuit layout with channel length/widths, it is possible to generate feasible process parameter combinations for silicon-germanium devices to explore the possible use of such devices in the circuit. Using the same electrical criteria as in Case 1, (Vd = 1 V, Vg = 1V,

<table>
<thead>
<tr>
<th>N_d(x10^16 cm^-3)</th>
<th>Bulk t_ox(nm)</th>
<th>Si-Ge t_ox(nm)</th>
<th>N_d(x10^16 cm^-3)</th>
<th>Bulk t_ox(nm)</th>
<th>Si-Ge t_ox(nm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>8.4</td>
<td>8.3</td>
<td>5.8</td>
<td>6.5</td>
<td>5.9</td>
</tr>
<tr>
<td>1.4</td>
<td>8.1</td>
<td>7.8</td>
<td>6.2</td>
<td>6.5</td>
<td>5.8</td>
</tr>
<tr>
<td>1.8</td>
<td>7.9</td>
<td>7.5</td>
<td>6.6</td>
<td>6.4</td>
<td>5.7</td>
</tr>
<tr>
<td>2.2</td>
<td>7.7</td>
<td>7.3</td>
<td>7.0</td>
<td>6.3</td>
<td>5.7</td>
</tr>
<tr>
<td>2.6</td>
<td>7.5</td>
<td>7.0</td>
<td>7.4</td>
<td>6.2</td>
<td>5.6</td>
</tr>
<tr>
<td>3.0</td>
<td>7.3</td>
<td>6.8</td>
<td>7.8</td>
<td>6.2</td>
<td>5.5</td>
</tr>
<tr>
<td>3.4</td>
<td>7.2</td>
<td>6.7</td>
<td>8.2</td>
<td>6.1</td>
<td>5.4</td>
</tr>
<tr>
<td>3.8</td>
<td>7.1</td>
<td>6.5</td>
<td>8.6</td>
<td>6.0</td>
<td>5.4</td>
</tr>
<tr>
<td>4.2</td>
<td>6.9</td>
<td>6.4</td>
<td>9.0</td>
<td>6.0</td>
<td>5.3</td>
</tr>
<tr>
<td>4.6</td>
<td>6.8</td>
<td>6.3</td>
<td>9.4</td>
<td>5.9</td>
<td>5.3</td>
</tr>
<tr>
<td>5.0</td>
<td>6.7</td>
<td>6.1</td>
<td>9.8</td>
<td>5.9</td>
<td>5.2</td>
</tr>
<tr>
<td>5.4</td>
<td>6.6</td>
<td>6.0</td>
<td>10.2</td>
<td>5.8</td>
<td>5.2</td>
</tr>
</tbody>
</table>
\( V_s = 0 \text{ V}, V_b = 0 \text{ V for } I_d = 200 \mu A \), it is possible to generate combinations of \( N_a-t_{ox} \) values, which are shown in Table 5.

The data generated show that for the same electrical criteria (and assuming all other parameter values to be the same), silicon-germanium devices require a smaller oxide thickness for the same doping levels. This information can be used in considering whether certain silicon-germanium device are feasible to fabricate for use in a given circuit.

Another problem which can be addressed is: given a set of electrical specifications, and a desirability to reduce certain parameter values (such as substrate doping and oxide thickness), what is the best combination of those parameter values which satisfies both conditions?

Clearly, to answer this question, it is necessary to quantify the “desirability” factor, which should be proportional to the tolerance of error in the satisfaction of the electrical specification; that is, reduction of both device parameter values may be possible, but only at the cost of straying from the desired electrical specifications. The selection of a desirability factor can be thought of as the selection of the values of \( \beta \) and \( \lambda \)’s in Equation 4 of Section 4.1.3.

An example optimization scenario was run, the downhill simplex routine was utilized for a penalty function formulated according to the device parameter penalty format given in Equation 4 of Section 4.1.3. The purpose of this scenario is to demonstrate that the optimization routines converge correctly to the proper values, and that through proper formulation of a penalty function (through judicious selection of the values of \( \beta \) and \( \lambda \)’s), it is possible to create trade-offs between the reduction of certain parameter values and the error in satisfaction of the specified electrical criteria.

Electrical criteria data were generated by the models using parameter values of \( N_a=3 \times 10^{16} \text{ cm}^{-3}, t_{ox}=8 \times 10^{-7} \text{ cm}, W=0.9 \mu m, L=0.45 \mu m, \text{ a front-gate doping of } 8 \times 10^{20} \text{ cm}^{-3}, \text{ and a junction depth of } 500 \text{ nm. Three data points were calculated for use in this optimization problem, as specified in Table 6.} \)
TABLE 6. Electrical Data used for Formulated Penalty Function

<table>
<thead>
<tr>
<th>$V_d$(V)</th>
<th>$V_g$(V)</th>
<th>$V_s$(V)</th>
<th>$V_b$(V)</th>
<th>$I_d$(µA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>0.5</td>
<td>0</td>
<td>330</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0.5</td>
<td>0</td>
<td>216</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
<td>0.5</td>
<td>0</td>
<td>161</td>
</tr>
</tbody>
</table>

The penalty function was constructed using these three data points, with all $\lambda$ values of Equation 4 equal to 1, and specified to be a function solely of oxide thickness and substrate doping. Parameter values were input logarithmically to allow for numbers roughly on the same scale (See Section 4.1.2 for discussion of scaling). Subsequent optimization runs added a weighted penalty function with all $\alpha$ values from Equation 3 of Section 4.1.2 equal to one and varying values for $\beta$. This is equivalent to varying the desirability factor: the larger the value of $\beta$, the larger tolerance for error in the satisfaction of the electrical criteria.

Since the final penalty function constructed had the potential for several local minima within its space, the best method to use for this function is the simplex method. The simplex optimization routine was started using an initial simplex of $(N_a, t_{ox})$ combinations of $(1 \times 10^{15}, 1 \times 10^{-7})$, $(1 \times 10^{15}, 1 \times 10^{-4})$, $(1 \times 10^{19}, 1 \times 10^{-7})$. Substrate doping values are in units of cm$^{-3}$ and oxide thickness values are in units of cm. The initial simplex was created by picking an arbitrary (but valid) point in the substrate doping-oxide thickness space and extending the point to the maximal upper bound in each parameter direction. The bounds for substrate doping were specified as $10^{14}$ to $10^{19}$ cm$^{-3}$, and $5 \times 10^{-7}$ to $10^{-4}$ cm for oxide thickness.

The optimization results are shown in Table 7.

TABLE 7. Optimization Results using Downhill Simplex Method

<table>
<thead>
<tr>
<th>$\beta$</th>
<th>Iteration count</th>
<th>$N_a$(x10$^{16}$ cm$^{-3}$)</th>
<th>$t_{ox}$(nm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>57</td>
<td>3.0</td>
<td>8.0</td>
</tr>
<tr>
<td>$10^{-12}$</td>
<td>56</td>
<td>3.0</td>
<td>8.0</td>
</tr>
<tr>
<td>$10^{-11}$</td>
<td>56</td>
<td>3.0</td>
<td>8.0</td>
</tr>
<tr>
<td>$2 \times 10^{-11}$</td>
<td>54</td>
<td>2.9</td>
<td>8.0</td>
</tr>
<tr>
<td>$4 \times 10^{-11}$</td>
<td>50</td>
<td>2.8</td>
<td>8.1</td>
</tr>
</tbody>
</table>
TABLE 7. Optimization Results using Downhill Simplex Method

<table>
<thead>
<tr>
<th>β</th>
<th>Iteration count</th>
<th>$N_a \times 10^{16} \text{ cm}^{-3}$</th>
<th>$t_{ox} \text{ (nm)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6 \times 10^{-11}$</td>
<td>50</td>
<td>2.7</td>
<td>8.2</td>
</tr>
<tr>
<td>$8 \times 10^{-11}$</td>
<td>59</td>
<td>2.5</td>
<td>8.2</td>
</tr>
<tr>
<td>$10^{-10}$</td>
<td>63</td>
<td>2.4</td>
<td>8.3</td>
</tr>
<tr>
<td>$10^{-9}$</td>
<td>94</td>
<td>0.01</td>
<td>11.5</td>
</tr>
<tr>
<td>$10^{-8}$</td>
<td>82</td>
<td>0.01</td>
<td>10.9</td>
</tr>
<tr>
<td>$10^{-7}$</td>
<td>107</td>
<td>0.01</td>
<td>7.8</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>126</td>
<td>0.01</td>
<td>5</td>
</tr>
<tr>
<td>$10^{-5}$</td>
<td>134</td>
<td>0.01</td>
<td>5</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>330</td>
<td>0.01</td>
<td>5</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>285</td>
<td>0.01</td>
<td>5</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>285</td>
<td>0.01</td>
<td>5</td>
</tr>
<tr>
<td>$10^{-1}$</td>
<td>285</td>
<td>0.01</td>
<td>5</td>
</tr>
</tbody>
</table>

As can be seen, the simplex routine utilizes more iterations to find the minimum penalty function value as the ratio of least-squares to weighted aspects of the penalty function become comparable in size. Secondly, by varying the value of $\beta$, it is possible to stress different aspects of the penalty function to accommodate different circumstances. Notice that, when the value of $\beta$ is 0, which is a penalty function which only stresses the satisfaction of electrical criteria, the optimization routine returns with the correct values which satisfy all three electrical criteria. As the value of $\beta$ increases, the effect of the inclusion of the weighted penalty function in the total penalty function becomes evident. Ultimately, for sufficiently large $\beta$, the penalty function stresses the weighted penalty function portion of the total penalty function, and the optimization routine converges to the parameter values which are the smallest possible.

From test cases performed using the different optimization routines, it was concluded that the downhill simplex method proved to be sufficiently fast and accurate enough to use for optimization of the model penalty functions which follow the format given in Section 4.1.3. The simulated annealing method also converges swiftly and accurately to a given point, but require a more complicated setup procedure since an annealing schedule must be selected. The gradient-based minimization had problems if the valid range of parameter values was narrow enough that the step sizes calculated by the routine
resulted in parameter value at or outside one of the boundaries of that parameter’s valid
range; when this occurs, the routine terminated at the parameter’s boundary value. Thus, it
was concluded that the downhill simplex method routine is best used for semiconductor
device parameter penalty functions with the format stated in Section 4.1.3.

If the format of the penalty function is altered (e.g., the weighted penalty function
is replaced by a function which penalizes parameter value deviations from specified val-
ues), then it may be possible to formulate penalty functions such that a different optimiza-
tion routine works best on a given function. For instance, if a given penalty function has an
analytical form of its gradient which can be implemented, the gradient-based minimiza-
tion may prove to be useful. Also, the format of the penalty function implemented in this
work does not have multiple local minima; if such a penalty function is implemented in
the future, then the simulated annealing routine, which works well for minimization of
functions with multiple local minima, may prove to be of use as well.

There is no single optimization routine which works well for all cases. The process
of selection of the optimization routine to use in the penalty function minimization must
consider the nature of the penalty function to be minimized. This consideration is espe-
cially important if new formats of the penalty function are created and implemented.
Chapter 7

Conclusions

A semiconductor device parameter space exploration tool has been presented here and has been illustrated in several cases which would be of use to circuit designers and device engineers. The implemented technique of rapid analytical modeling can tighten circuit and device design loops and eliminate some of the time-consuming numerical simulations which commonly exist in current circuit and device design phases.

Four potential test scenarios have been presented, each of which shows how the use of this tool provides information which can be used in the decision making process which occurs during device technology and device parameter selection.

7.1 Future Research Work

There are several limitations on the capabilities of the current system. The system was originally intended to analyze static performance of four terminal semiconductor devices. As implemented, the system only can deal with four terminal devices with one output quantity (i.e. voltage biases on all terminals, and current as the output quantity). However, the infrastructure is general enough to adapt the model protocol to allow for any number of input terminals and any number of outputs; this will lead to more complex bookkeeping and processing of information. This modification might be useful for analy-
sis of bipolar transistors, which only have three terminals, or for circuit analysis where multiple output quantities are important.

Another limitation of the system is that since it was originally designed to analyze static performance of devices, any dynamic performance cannot be evaluated. Therefore, any time related analysis cannot be performed using the system. While the model protocol can be changed to include methods for specifying time-varying behavior of a certain device, it may be necessary to modify the system infrastructure to allow for proper analysis of this information.

There are other improvements which can be made. Periodically, as newer models for semiconductor devices are developed (and other models become obsolete), the model definitions file for the tool can be modified accordingly as described in Chapter 5. Incorporation of a fitting parameter extraction program into the tool may expedite the integration of newer models.

There are also several improvements which can be made to the system which are more cosmetic in nature. The current plotting tool only allows for displays of two variables. A three dimensional surface plotting program, similar to MATLAB's SURF routine, would allow for visual presentation of device parameter space exploration of three parameters, which could also be of use. A graphical interface to allow for more user-friendly specification of a penalty function for the optimization tool can be implemented, although due to the general nature of the penalty function, it may be difficult to provide enough flexibility to allow for complete specification.

7.2 Incorporation into the CAD environment

It is envisioned that this tool would be useful as a complement to, rather than a replacement for, a circuit simulation tool such as SPICE or a device simulator such as MEDICI. Using a rapid analytical modeling approach implemented by this tool allows for swift computation of initial guesses for semiconductor device parameters, which can then be verified and further specified by SPICE, MEDICI, or other simulators. Use of this tool thus can cut down on the decision time required to select feasible device parameters.
Ideally, the ultimate goal would be the integration of this tool with SPICE and other circuit design tools to allow for a streamlined device technology and device parameter selection procedure. For instance, in a given circuit, if all of the electrical specifications are known, the tool described in this thesis could generate feasible parameter sets for process parameters which could be analyzed to determine which set most closely matches available processes. Once a process is selected, the circuit simulation tool can be used to more accurately tune the layout parameters using more accurate device models.

The technique of semiconductor device parameter exploration can be used to answer a variety of questions for circuit design engineers. The tool described in this thesis demonstrates the basic techniques of parameter space exploration for the purposes of analyzing static behavior of four-terminal semiconductor devices. This type of analysis provides information which can be useful in the selection of a device technology and device parameters for a given circuit design. Such a tool can be a useful contributor to the circuit design process and thus should be included as a part of future distributed CAD environments.
REFERENCES


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title/Authors</th>
<th>Details</th>
</tr>
</thead>
</table>
Appendix A: Optimization Algorithms

This appendix describes in detail the optimization algorithms implemented for use with this tool. There are four optimization routines which were implemented. One of the routines, the exhaustive search routine, is straightforward and requires no further explanation. The other three routines are numerical algorithms which were taken from Numerical Recipes in C [PRESS] and implemented in the Java language. Description of the algorithms were also taken from [PRESS].

Downhill Simplex

The downhill simplex method, first derived by Nelder and Mead [NELDER] requires only function evaluations (but not function derivatives). It is not very efficient in terms of the number of function evaluations it requires. However, this method is frequently a good method to use if the computational evaluation time for the function involved is relatively small.

This method has a geometrical naturalness which makes it simple to describe. A simplex, is the geometrical figure consisting, in N dimensions, of N + 1 points (or vertices) and all their interconnecting line segments, polygonal faces, etc. In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron, though not necessarily a regular tetrahedron. (This method is not related to the simplex method of linear programming, which makes use of the geometrical concept of a simplex, but is otherwise completely unrelated to this algorithm). In general the only simplexes of interest are the nondegenerate simplexes (i.e., the ones that enclose a finite inner N-dimensional volume. If a point of a nondegenerate simplex is taken as the origin, the other N points serve as vector directions that span the N-dimensional vector space.

For multidimensional minimization, the goal is to give this algorithm a starting guess, an N-dimensional vector as the first point to try. The algorithm then makes its own
way downhill through the vast complexity of N-dimensional topography until it encounters a minimum.

The downhill simplex method must be started not just with a single point, but with N + 1 points, defining an initial simplex. This initial simplex can be generated by simply taking an initial point \( P_0 \) and using Equation A1 to generate the other N points:

\[
P_i = P_0 + \lambda e_i
\]  
(EQ. A1)

In Equation A1, \( e_i \) is the \( i \)th unit vector in one of the N dimensions, and \( \lambda \) is a constant which reflects a guess of the problem’s characteristic length scale.

The downhill simplex method now takes a series of iterations; most iterations simply move the point of the simplex where the function is largest ("highest point") through the opposite face of the simplex to a lower point. These iterations are called reflections, and they are constructed to conserve the volume of the simplex, and thus maintain its non-degeneracy. When it can do so, the algorithm expands the simplex in one or another direction to take larger reflections. When it reaches a "valley floor", the algorithm contracts itself in the transverse direction and tries to "ooze" down the valley. If there is a situation where the simplex is trying to "pass through the eye of a needle," it contracts itself in all directions, pulling itself in around its lowest point.

![Possible Iterations of the Simplex Algorithm](image-url)

Figure A1: Possible Iterations of the Simplex Algorithm
Figure A1(a) shows an initial simplex. Figure A1(b) shows the result after a simple reflection. Figure A1(c) shows the result after a reflection and expansion. Figure A1(d) shows the result after a one-dimensional contraction. Figure A1(e) shows the result after a multidimensional contraction.

Termination criteria can be delicate in any multidimensional minimization routine. It is possible to identify one iteration of this simplex algorithm. It is then possible to terminate when the vector distance moved in a given iteration is fractionally smaller in magnitude than some user-specified tolerance. Once this tolerance is reached, the routine may end and return the final simplex, which should be contracted around the point where the function is at a minimum.

**Conjugate Gradient**

The conjugate gradient method assumes the existence of a gradient function which calculates a vector of the first partial derivatives of a given function. In using this extra information it is possible to reduce, by as much as an order of magnitude, the computation time and iterations require to compute the minimum of a function. This method is only useful for forms that can be approximated by a quadratic function:

\[
f(x) = c \cdot b \cdot x + (1/2) x \cdot A \cdot x \tag{EQ. A2}
\]

where the number of unknown parameters in \( f \) is the number of free parameters in \( A \) and \( b \). Note that since this method only works well with functions that can be approximated by quadratic forms. It may have problems with functions with multiple local minima scattered among a global minimum, since it tends to converge to the nearest local minimum.

The conjugate gradient technique is one of the various techniques which are characterized by a series of line minimizations. Line minimizations can be thought of as the following problem: Given as input the vectors \( P \) and \( n \), and the function \( f \), find the scalar \( \lambda \) that minimizes \( f(P + \lambda n) \), and replace \( P \) by \( (P + \lambda n) \), and \( n \) by \( \lambda n \). Gradient information is
not necessary in a line minimization algorithm, but such information can be used to intelligently pick the direction of minimization.

Such a choice of directions is more difficult than first appears. The obvious selection of the direction of steepest descent may lead to needless iterations in certain situations. For instance, take the case as illustrated in Figure A2. The curves shown are contour lines of a given function. If a function has a long, narrow valley, it is possible that a successive line minimization routine which takes the direction of steepest descent will require many iterations to reach the bottom of the valley. This is due to the fact that using the method of steepest descent requires successive minimizations in directions perpendicular to the last travelled direction (since the steepest direction is the direction of the local downhill gradient; at the minimum along a line, the new gradient is always perpendicular to the direction just travelled).

![Figure A2: Inefficiency of “Steepest Descent” Method](image)

The conjugate gradient method constructs a new direction to perform line minimizations which is conjugate to the old gradient, and conjugate to all previous directions travelled, if possible. Starting with an arbitrary initial vector $g_0$ and letting $h_0 = g_0$, the conjugate gradient method constructs two sequences of vectors from the recurrence

$$
g_{i+1} = g_i - \lambda_i A \cdot h_i$$
$$
h_{i+1} = g_{i+1} + \gamma_i h_i$$

$i=0,1,2,...$  \hspace{1cm} (EQ. A3)

where $A$ is the Hessian matrix.
The generated vectors satisfy the orthogonality and conjugacy conditions:

\[ g_i \cdot g_j = 0 \quad h_i \cdot A \cdot h_j = 0 \quad g_i \cdot h_j = 0 \quad i \neq j \]  

(EQ. A4)

The problem with Equation A3 is that the Hessian matrix \( A \) is not known. To get around this problem it is necessary to use the following theorem: Given the gradient information \( g_i = - \nabla f(P_i) \) for some point \( P_i \) where \( f \) is of the form in Equation A2, proceed from \( P_i \) along the direction \( h_i \) to the local minimum of \( f \) located at some point \( P_{i+1} \). It is possible to set \( g_{i+1} = - \nabla f(P_{i+1}) \). Then, this \( g_{i+1} \) is the same vector that would have been constructed by equation A3, without requiring knowledge of \( A \). (For the full proof of this see [PRESS, pp.414]).

Thus, it is possible to build an algorithm which requires no knowledge of the Hessian matrix. A sequence of directions \( h_i \) is constructed, using only line minimizations, evaluations of the gradient vector, and an auxiliary vector to store the latest in the sequence of \( g \)'s.

Polak and Ribiere introduced a tiny variation in the algorithm. They proposed using the form [POLAK]:

\[ \gamma_i = ((g_{i+1} \cdot g_i) \cdot g_{i+1})/(g_i \cdot g_i) \]  

(EQ. A5)

which improves upon the performance of the original algorithm by taking into account the fact that most functions are not exact quadratic forms, but are only approximations of such forms of equations.

**Simulated Annealing**

At the heart of the method of simulated annealing is an analogy with thermodynamics, specifically with the way that liquids freeze and crystallize, or metals cool and anneal. At high temperatures, the molecules of a liquid move freely with respect to one another. If the liquid is cooled slowly, thermal mobility is lost. The atoms can line them-
selves up and form a pure crystal that is completely ordered. This crystal is the state of minimum energy for this system. The amazing fact is that, for slowly cooled systems, nature is able to find this minimum energy state. If liquid metal is cooled quickly, it does not reach this state but rather ends up in a polycrystalline or amorphous state of higher energy.

The basic ideas of simulated annealing are applicable to minimization problems where the object is to find the minimum of a given function which has many local minima. There are four requirements to implement a simulated annealing algorithm: an objective function (e.g., a penalty function), a starting point \( x \), a control parameter \( T \) (which represents temperature in the liquid metal analogy) which starts at a positive number and is slowly reduced each iteration according to some arbitrary annealing schedule, and a generator of random changes which moves a point \( x \) to a random new point \( x' \).

The method of random change generation used in this work is a modification of the downhill simplex method. This amounts to replacing the single point \( x \) by a simplex of \( N + 1 \) points. Iterations of the annealing algorithm are simply reflections, expansions, and contractions of the simplex. The modification is this: a positive, logarithmically distributed random variable, proportional to \( T \), is added to the stored function value associated with each vertex of the simplex, and a similar random value is subtracted from the function value of each point attempted as a replacement. This method always accepts a true downhill step, but will sometimes accept an uphill one. As \( T \) approaches 0, the algorithm reduces to the normal simplex routine, and converges to a local minimum.

At a finite value of \( T \), the simplex expands to a scale that approximates the size of the region that can be reached at that value, and then executes a stochastic, tumbling Brownian motion within that region, sampling new, approximately random points as it does so. If the temperature is reduced sufficiently slowly, it becomes highly probable that the simplex will shrink to the region containing the lowest global minimum encountered.

Termination criteria for a simulated annealing algorithm can be determined in two ways: First, if successive iterations result in a fractional convergence tolerance of function values which is less than a specified tolerance, then the algorithm terminates. However, the
algorithm will also terminate at the conclusion of the annealing schedule, after the temperature variable has been reduced to 0.

**Test Case**

The three optimization routines were run using the following penalty function:

\[
f(x_1, x_2) = (x_1 - 2)^2 + (x_2 - 3)^2
\]

This function has a single minimum at (2,3). For the simplex and annealing routines, the initial simplex given was \{ {25,23}, {25,1e99}, {1e99,23} \}. A gradient function was not supplied for the conjugate gradient method; all gradient information was computed numerically. The annealing routine started at an annealing temperature of 15 and is reduced by 1 degree after each run. Each run involved 100 random changes to the initial simplex.

Below lists the results of the routines. In all three cases, the routines successfully converge to the correct minimum of (2,3).

<table>
<thead>
<tr>
<th>Routine</th>
<th>Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplex</td>
<td>570</td>
</tr>
<tr>
<td>Conjugate Gradient</td>
<td>600</td>
</tr>
<tr>
<td>Simulated Annealing</td>
<td>810</td>
</tr>
</tbody>
</table>

All of the routines converged fairly quickly to the desired minimum. The simplex method required 1434 function evaluations to reach the minimum; the annealing method required 834, and also had the extra computational burden of calculating random changes for simplex points. The conjugate gradient method required only 3 line minimizations to find the desired minimum; however, each line minimization required many function evaluations. This, along with the extra computation time needed to compute the gradient values, explains the comparable execution times.
Appendix B: Model Derivations

There are four models which have been implemented for use with this tool: a region-based bulk CMOS model, a surface-potential based bulk CMOS model, a surface-potential based Silicon-on-Insulator with Active Substrate (SOIAS) model, and a silicon-germanium model. This appendix shows the derivations for the model equations which were later implemented in computer code.

Region-based bulk CMOS

The region-based bulk CMOS model can be found in any classic electronics textbook such as [HOWE]. It is the basic model which is simply a first-order approximation of the operation of a bulk NMOS transistor. This model considers only the drift component of drain current. There are three regions of operation: cutoff, triode, and saturation. The cutoff region occurs when the gate-to-source voltage, $V_{GS}$, is less than the threshold voltage, $V_T$. In this region it is assumed that the current is zero.

If $V_{GS}$ is greater than $V_T$, then current begins to flow. The relationship between drain current and channel charge is:

$$I_D = -Wv_y(y)Q_N(y) \quad \text{(EQ. A7)}$$

where $W$ represents the channel length, $v_y$ represents the drift velocity, and $Q_N$ is the channel charge. Equations for the drift velocity and channel charge are given by:

$$v_y(y) = -\mu_n E_y = \mu_n (dV/dy) \quad \text{(EQ. A8)}$$

$$Q_N(y) = -C_{ox}(V_{GB} - V(y) - V_T) \quad \text{(EQ. A9)}$$

where $\mu_n$ is the carrier mobility (here, of electrons), $E_y$ is the electric field (which equals the derivative of voltage with respect to distance), $C_{ox}$ is the oxide capacitance per unit area, $V_{GB}$ is the gate-to-bulk voltage, and $V_T$ is the threshold voltage.
Substituting these two equations into Equation A7 and rearranging leads to the equation (assume the source of the device is grounded):

\[ I_{Ddx} = \mu_n C_{ox} W (V_{GS} - V_{VT}) dV \]  

(EQ. A10)

Integration of Equation A10 leads to the current equation for the triode region:

\[ I_D = (1/2) \mu_n C_{ox} (W/L) (2(V_{GS} - V_{VT}) V_{DS} - V_{DS}^2) \]  

(EQ. A11)

As the value of the drain-source voltage is increased, the channel begins to exhibit an effect known as pinch-off, where increasing the drain-source voltage no longer increases the drain current. At this point the CMOS device enters the saturation region, and the current is constant. The equation for the saturation region can be derived by substituting for \( V_{DS} \) in Equation A11. The saturation region begins when \( V_{DS} = V_{GS} - V_{VT} \):

\[ I_D = (1/2) \mu_n C_{ox} (W/L) (V_{GS} - V_{VT})^2 \]  

(EQ. A12)

Thus, the final set of equations is:

\[
I_{ds} = \begin{cases} 
0 & V_{gs} < V_{th} \\
\frac{1}{2} \mu C_{ox} \frac{W}{L} (2(V_{gs} - V_{th}) V_{ds} - V_{ds}^2) & V_{ds} \leq V_{gs} - V_{th} \\
\frac{1}{2} \mu C_{ox} \frac{W}{L} (V_{gs} - V_{th})^2 & V_{ds} > V_{gs} - V_{th}
\end{cases}
\]

Surface Potential based bulk CMOS

The surface potential based (charge sheet) model considers both the drift and diffusion components of drain current. The equation for drain current is one equation which is valid for all regions of operation of the transistor. It is a considerably more complicated model than the region-based model, and incorporates several second-order effects which allow it to better mimic the operation of an actual device [TURCHETTI]. The derivation given in this section is taken from [TURCHETTI] and [ARORA].
An expression for the surface potential can be derived from the Poisson equation, assuming a Boltzmann distribution of carriers:

\[
\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = (qN_a/\varepsilon_{si})(\exp((\phi-2\phi_F-\phi_n)/\phi_t) - \exp(-(\phi-V_b)/\phi_t) + 1) \quad (\text{EQ. A13})
\]

where \(\phi\) is the electrostatic potential, \(q\) is electron charge, \(N_a\) is substrate doping, \(\varepsilon_{si}\) is the permittivity of silicon, \(\phi_n\) is the quasi-fermi potential, \(V_b\) is the applied voltage on the bulk terminal, \(\phi_t\) is the thermal voltage, and \(\phi_F\) is the bulk fermi potential.

Also, using Gauss’ law, the field at the silicon surface is given as:

\[
\frac{\partial \phi_s}{\partial x} = (C_{ox}/\varepsilon_{si})(V_g-V_{fb}-\phi_s) \quad (\text{EQ. A14})
\]

where \(C_{ox}\) is the gate oxide capacitance per unit area, and \(V_{fb}\) is the flatband voltage of the transistor.

Following the work of [TURCHETTI], it has been assumed that the second derivative of potential in the y direction is a constant term. Integration of Equation A13 in the x direction from the source (\(\phi=\phi_g\)) to the bulk (\(\phi=V_b\)), and substitution of Equation A14 into the result, leads to:

\[
\Gamma - \phi_t(\exp(-\phi_s/\phi_t)-1) - \phi_t\exp(-(2\phi_F+\phi_n)/\phi_t)(\exp(\phi_s/\phi_t) - 1) - f_s\phi_s = 0 \quad (\text{EQ. A15})
\]

where

\[
\Gamma = (1/\gamma^2)(V_{gb}-V_{fb}-\phi_s)^2 \quad (\text{EQ. A16})
\]

\[
f_s = 1 - (\varepsilon_{si}/qN_a)(\partial^2 \phi_s/\partial y^2) \quad (\text{EQ. A17})
\]

\[
\gamma = \sqrt{(2\varepsilon_{si}qN_a)/C_{ox}} \quad (\text{EQ. A18})
\]

The general solution for Equation A15 can only be solved through numerical iterations using Newton’s method. The iterative updates are given by:

\[
\Delta \phi_s = (f/f^\gamma)/( (ff^\gamma)/(2f^2 - 1) - 1) \quad (\text{EQ. A19})
\]
where \( f \) is the left hand side of Equation A15, and \( f' \) and \( f'' \) are its first and second derivatives. Starting with an initial guess, the guess is updated until Equation A15 is solved to a desired error tolerance. The initial guess is given by:

\[
\phi_s = 2\phi_f - \phi_{bs} + \phi_t \ln \left[ 1 + (1/\phi_t) \left( \frac{(U_g - 2\phi_f + \phi_{bs})^2}{\phi_f - \phi_{bs}} \right) \right] \quad \text{(Inversion)} \quad \text{(EQ. A20)}
\]

\[
\phi_s = \phi_{sat} \quad \text{(Depletion)} \quad \text{(EQ. A21)}
\]

\[
\phi_s = -\phi_t \ln \left( 1 + \frac{U_g^2}{(\phi_t)^2} \right) \quad \text{(Accumulation)} \quad \text{(EQ. A22)}
\]

\[
U_g = V_{gb} - V_{fb} \quad \text{(EQ. A23)}
\]

\[
\phi_{sat} = U_g + \left( \frac{\gamma^2}{2} \right) \left[ 1 - \left( 1 + 4\left( U_g \phi_t / \gamma^2 \right) \right) \right] \quad \text{(EQ. A24)}
\]

Using this method it is possible to solve for the surface potentials on the source side, \( \phi_{ss} \).

From Equation A14, using the body as reference, the gate charge per unit area is given as:

\[
Q_g = C_{ox} (V_{gb} - V_{fb} - \phi_s) \quad \text{(EQ. A25)}
\]

Using a standard depletion approximation, and taking into account the lateral field gradient, the body charge per unit area is:

\[
Q_b = -\gamma C_{ox} \sqrt{(f_s \phi_s - \phi_t)} \quad \text{(EQ. A26)}
\]

The channel charge can be derived from the charge neutrality condition:

\[
Q_i = -Q_g - Q_b = -C_{ox} (V_{gb} - V_{fb} - \phi_s - \phi_t) \gamma (\phi_s - \phi_t) \quad \text{(EQ. A27)}
\]

\[
\gamma_e = \frac{1 + f_s}{2} \quad \text{(EQ. A28)}
\]

where the \( f_s \) term has been taken out of the square root of Equation A26 using a standard square root approximation.
The square root term can then be approximated by a linear function:

\[ \sqrt{(\phi_s - \phi_t)} \equiv \sqrt{(\phi_{ss} - \phi_t)} + \delta(\phi_s - \phi_{ss}) \]  \hspace{1cm} (EQ. A29)

\[ \delta = (1 - 1/(1.43 + 1.41 (\phi_{ss} - \phi_t) )/2 (\phi_{ss} - \phi_t)) \]  \hspace{1cm} (EQ. A30)

which can then be replaced in Equation A26 (modified to include the square root approximation) to yield:

\[ Q_i = - C_{ox}(V1 - \alpha(\phi_s - \phi_{ss})) \]  \hspace{1cm} (EQ. A31)

\[ V_1 = V_{gb} - V_{fb} - \phi_{ss} - \gamma_e(\phi_{ss} - \phi_t) \]  \hspace{1cm} (EQ. A32)

\[ \alpha = 1 + \gamma_e \delta \]  \hspace{1cm} (EQ. A33)

Rather than use a constant value mobility, the following approximation can be made:

\[ \mu = \mu_s/(1+\delta_0 (\mu_s/v_{sat})(d\phi_y/dy)) \]  \hspace{1cm} (EQ. A34)

\[ \mu_s = \mu_0/(1 + \gamma_e \sqrt{(\phi_{ss} - \phi_t)}/E_0 )^\gamma \]  \hspace{1cm} (EQ. A35)

\[ \delta_0 = K/(1.5 + K) \]  \hspace{1cm} (EQ. A36)

\[ K = (\mu_s/v_{sat})(\phi_{sd} - \phi_{ss})/L_{eff} \]  \hspace{1cm} (EQ. A37)

where \( \mu_0 \) is the zero field mobility, \( v_{sat} \) is the carrier saturation velocity, \( E_0 \) and \( \gamma \) are fitting parameters (set to 6.7 x 10\(^7\) and 1.6), \( \phi_{sd} \) and \( \phi_{ss} \) are the surface potentials at the drain and source side, and \( L_{eff} \) is the effective channel length.

Within the context of the charge-sheet approximation, the drain current equation can be derived by using the one-dimensional current continuity equation for electrons,

\[ I_{ds} = -W_{eff}\mu \left( Q_i(d\phi_y/dy) + \phi_t(dQ_i/dy) \right) \]  \hspace{1cm} (EQ. A38)
It is possible to replace Equation A31 and A34 in Equation A38 and integrate to get:

\[ I_{ds} = \left( \frac{C_{ox} W_{eff} \mu_s}{L_{eff} + h V_{dsx}} \right) [V_0 - (\alpha/2)V_{dsx}] V_{dsx} \]  

(EQ. A39)

\[ V_0 = V_1 + \alpha \phi_s \]  

(EQ. A40)

\[ h = \delta_0 (\mu_s / V_{sat}) \]  

(EQ. A41)

where \( V_{dsx} = \phi_{sd} - \phi_{ss} \). It is possible to use an empirical smoothing function to calculate \( V_{dsx} \):

\[ V_{dsx} = V_{dsat} \left[ 1 - \frac{\ln(1+\exp(A(1- (V_{ds}/V_{dsat}))))}{\ln(1+e^A)} \right] \]  

(EQ. A42)

\[ V_{dsat} = \frac{V_1}{\alpha} \]  

(EQ. A43)

where \( A \) is a fitting parameter (set to 10), and \( V_{dsat} \) represents the saturation voltage. The saturation voltage is the drain bias required to result in a drain current equal to the saturation channel current. Equation A43 is a good approximation, and can be improved upon by iterative techniques if desired.

The model also incorporates channel length modulation and series resistance effects. The fraction of the channel having velocity saturated carriers is given by:

\[ I_d = \sqrt{\left( \frac{\varepsilon_s \varepsilon_{ox} X_j}{\varepsilon_{ox}} \right) \ln(1 + \lambda(V_{ds} - V_{dsx}) \right)} \]  

(EQ. A44)

which can be subtracted from the effective channel length \( L_{eff} \). Series resistance can also be incorporated into Equation A39 to yield the final drain current equation:

\[ I_{ds} = \beta \left[ (V_0 - (\alpha/2)V_{ds}) V_{ds} \right] \left( 1 + \beta R_t[V_0 + (s-\alpha)V_{ds}] \right) \]  

(EQ. A45)

where

\[ \beta = \left( \frac{C_{ox} W_{eff} \mu_s}{L_{eff} + h V_{dsx}} \right) \]  

(EQ. A46)

\[ s = r[1 + \gamma_e/(2 (\phi_{ss} - \phi_s))] \]  

(EQ. A47)
\[
\begin{align*}
  r &= \left(\frac{q_i}{2}\right)\phi_t + q_i + 2\phi_t(V_{gb} - V_{fb} - \phi_{ss})/V_c^2) \quad \text{(EQ. A48)} \\
  q_i &= (V_{gb} - V_{fb} - \phi_{ss}/V_c)^2 - \phi_{ss} - \phi_t \quad \text{(EQ. A49)}
\end{align*}
\]

**Surface Potential based SOIAS**

The surface potential based SOIAS model is a physical, scalable, compact model developed by [NAREN]. The derivation presented here is taken from [NAREN].

![SOIAS device information](image)

**Figure A3: SOIAS device information**

The change in surface potential along the x axis is given by Poisson’s equation:

\[
\frac{\partial^2 \phi}{\partial x^2} = \left(\frac{qN_a/\varepsilon_{si}}{\exp((\phi - 2\phi_F)/\phi_t) - \exp((\phi_F - \phi)/\phi_t) + 1}\right) \quad \text{(EQ. A50)}
\]

where \(\phi\) is the electrostatic potential, \(q\) is the electron charge, \(N_a\) is the silicon film doping, \(\phi_F\) is the Fermi potential, and \(\phi_t\) is the thermal voltage. The following boundary conditions are used in solving Equation A50:
\[ \phi(x = 0) = \phi_s \quad \text{(EQ. A51)} \]
\[ \phi(x = t_{si}) = \phi_b \quad \text{(EQ. A52)} \]
\[ \phi'(x = 0) = (\varepsilon_{ox}/\varepsilon_{si})(\phi_s - V_g + V_{fbf})/t_{ox} \quad \text{(EQ. A53)} \]
\[ \phi'(x = t_{si}) = (\varepsilon_{ox}/\varepsilon_{si})(V_b - V_{fbb} - \phi_b)/t_{box} \quad \text{(EQ. A54)} \]
\[ V_{fbf} = -(\phi_t \ln(N_{fg}/n_i) + \phi_F) \quad \text{(EQ. A55)} \]
\[ V_{fbb} = -(\phi_t \ln(N_{ab}/n_i) + \phi_F) \quad \text{(EQ. A56)} \]

where \( N_{ab} \) is the doping of the back gate, \( N_{fg} \) is the doping of the front gate, and \( n_i \) is the intrinsic carrier concentration.

Integrating Equation A50 with the given boundary conditions gives:

\[ \frac{1}{2} \left[ (\phi_s - V_g')^2 - r_t^2 (V_b' - \phi_b')^2 \right] - k_1[k_3-k_4] - k_2[(k_4-k_3)/(k_3k_4)] - \phi_s - \phi_b = 0 \quad \text{(EQ. A57)} \]

where

\[ V_{g'} = V_g - V_{fbf} \quad \text{(EQ. A58)} \]
\[ V_{b'} = V_b - V_{fbb} \quad \text{(EQ. A59)} \]
\[ k_1 = \phi_t \exp(-2\phi_F/\phi_t) \quad k_2 = \phi_t \exp(\phi_F/\phi_t) \quad \text{(EQ. A60)} \]
\[ k_3 = \exp(\phi_s/\phi_t) \quad k_4 = \exp(\phi_b/\phi_t) \quad \text{(EQ. A61)} \]
\[ \gamma = \sqrt{2qN_a\varepsilon_{si}/C_{ox}} \quad \text{(EQ. A62)} \]
\[ r_t = (t_{ox}/t_{box}) \quad C_{ox} = (\varepsilon_{ox}/t_{ox}) \quad C_{box} = (\varepsilon_{ox}/t_{box}) \quad C_{si} = (\varepsilon_{si}/t_{si}) \quad \text{(EQ. A63)} \]

Equation A57 has two unknowns \( \phi_s \) and \( \phi_b \). For a given bias a second equation relating these surface potentials is required to determine their values. This second equation is derived from the fact that the surface potentials are related by the potential drop across
the fully depleted silicon film which is a function of the doping, dimensions of the structure, and the bias:

$$\phi_b = \phi_s - \frac{(qN_{a\text{Si}})^2}{(2\varepsilon_{\text{Si}})} + \frac{(C_{\text{box}}/C_{\text{ox}})V_b'}{2E_{\text{si}}}$$  \hspace{1cm} (EQ. A64)

The surface potential is then solved using a second order Newton’s iterative technique similar to that method explained in the derivation of the bulk surface potential model, substituting the appropriate surface potential equation. The surface potentials can be used to solve for the charge in the front and back channels:

$$Q_g = C_{\text{ox}}(V_g' - \phi_s) \quad Q_{bg} = C_{\text{box}}(V_b' - \phi_b)$$  \hspace{1cm} (EQ. A65)

Since SOIAS is a fully depleted device the charge in the body is dependent on the doping and film thickness:

$$Q_{b0} = -qN_{a\text{Si}}$$  \hspace{1cm} (EQ. A66)

By charge conservation the inversion charge will be:

$$Q_i = -[Q_g + Q_{bg} + Q_{b0}]$$  \hspace{1cm} (EQ. A67)

$$Q_i = -C_{\text{ox}}[V_g' - \phi_s + r_1(V_b' - \phi_b) - (qN_{a\text{Si}})/C_{\text{ox}}]$$  \hspace{1cm} (EQ. A68)

Using the charge sheet model approach, drain current can be written as:

$$I_{ds} = -\mu_{\text{eff}}[Q_i(d\phi_s/dy) - \phi_i(dQ_i/dy)]$$  \hspace{1cm} (EQ. A69)

which incorporates both the drift and diffusion components of drain current.

The following equations are used to determine the mobility in Equation A69:

$$\mu = (\mu_s)(1 + \delta_0(\mu_s/\mu_{\text{sat}})(d\phi_s/dy))$$  \hspace{1cm} (EQ. A70)

$$\mu_s = \mu_0 \left(1 + \left(\left(1/2\right)Q_i + Q_{b0}/E_0\right)^{\gamma}\right)$$  \hspace{1cm} (EQ. A71)

$$\delta_0 = K/(1.5 + K)$$  \hspace{1cm} (EQ. A72)
\[ K = \frac{\mu_0/v_{sat}}{(\phi_{sd} - \phi_{ss})/L_{eff}} \quad \text{(EQ. A73)} \]

where \( \mu_0 \) is the zero field mobility, \( v_{sat} \) is the saturation velocity, and \( L_{eff} \) is the effective channel length. \( E_0 \) and \( v \) are fitting parameters which are set to \( 6.7 \times 10^7 \) and 1.6 for electrons.

Substituting into Equation A69, and integrating:

\[ I_{ds} = \frac{\mu_s C_{ox} W_{eff}}{(L_{eff} + hV_{dsx})[V_0 - (1/2)(V_{dsx} + 2\phi_{ss})]V_{dsx}} \quad \text{(EQ. A74)} \]

\[ h = \delta_0(\mu_s/v_{sat}) \quad \text{(EQ. A75)} \]

\[ V_0 = V_{gs'} + r_t(V_{bs'} - \phi_b) - (qN_{d'si})/C_{ox} + \phi_t \quad \text{(EQ. A76)} \]

\[ V_{gs'} = V_g - V_s \quad V_{bs'} = V_b - V_s \quad \text{(EQ. A77)} \]

where \( V_{dsx} \) is a smoothing function given by:

\[ V_{dsx} = V_{dsat}[1 - (\ln(1+\exp(A(1 - (V_{ds}/V_{dsat}))))/(\ln(1+e^A)))] \quad \text{(EQ. A78)} \]

\[ V_{dsat} = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \quad \text{(EQ. A79)} \]

\[ a = (1/2)(\mu_s/v_{sat}) - 2h \quad b = hV_2 - 2L_{eff} - V_3\mu_s/v_{sat} \quad c = L_{eff}V_2 \quad \text{(EQ. A80)} \]

\[ V_2 = V_0 - \phi_t - \phi_s \quad V_3 = V_2 - \phi_t \quad \text{(EQ. A81)} \]

\( V_{dsat} \) represents the voltage where the device enters the saturation region. It is obtained analytically by equating Equation A74 with \( V_{dsx} = V_{dsat} \) to be the drain saturation current. Note: Equation A80 will be modified later to include the effects of source-drain series resistance.

The model also includes a modification for channel length modulation:

\[ I_d = \sqrt{\varepsilon_{si} t_{si} C_{ox}} \ln(1 + \lambda(V_{ds} - V_{dsx})) \quad \text{(EQ. A82)} \]

which is subtracted from the calculated effective channel length.
The effect of source drain series resistance is captured by correcting the drain current equation to give:

$$I_{ds} = \frac{I_{ds0}}{[1 + (R_{sd}/W)(I_{ds0}/V_{dsx})]} \tag{EQ. A83}$$

where $I_{ds0}$ is the drain current calculated in Equation A74 (with the modification to the calculation of $V_{dsx}$, which depends on $V_{dsat}$, described below).

It is then important to modify the equations for $V_{dsat}$, in order to reflect the new equation for current. Thus Equation A80 should be replaced with the new coefficients as follows:

$$a = (1/2)(\mu_s/v_{sat}) - 2h + 2R_{sd} \tag{EQ. A84}$$

$$b = hV_2 - 2L_{eff} - V_3\mu_s/v_{sat} - R_{sd}(4V_3 - V_2) \tag{EQ. A85}$$

$$c = L_{eff}V_2 + 2R_{sd}V_2V_3 \tag{EQ. A86}$$

The final effect which is included is the short channel effect of drain induced barrier lowering (DIBL) which is captured by a shift in $V_{gs}$ as follows:

$$\Delta V_{gs} = \left(C_{box}/C_{ox}\right)[(\Phi_1 + \Phi_2)/\Phi_3] \tag{EQ. A87}$$

$$\Phi_1 = 2\Phi_4 + (V_{ds} + \Phi_2)(1 - \exp(L_{eff}/l)) \tag{EQ. A88}$$

$$\Phi_2 = 2[\Phi_4^2 + \Phi_4(V_{ds} + \Phi_4)(\exp(L_{eff}/l) - 1)] \tag{EQ. A89}$$

$$\Phi_3 = 4\sinh^2(L_{eff}/(2l)) \tag{EQ. A90}$$

$$\Phi_4 = \phi_{sd} - \phi_{b0} - \phi_s \tag{EQ. A91}$$

$$C_{box} = (C_{box}C_{si})/(C_{box} + C_{si}) \quad C_{ox} = (1/C_{ox}) + (1/C_{si}) \tag{EQ. A92}$$

$$1 = \sqrt{[(\epsilon_{si}C_{ox})/(\zeta C_{ox}C_{box})]} \tag{EQ. A93}$$

where $\zeta$ is a fitting parameter.
Silicon-Germanium

The silicon germanium model was implemented by modifying the bulk surface potential based model to include the effects of the buried silicon-germanium channel [SADEK]. The derivation is based on the concept that a silicon-germanium device behaves basically like a bulk CMOS device except for two effects: first, the mobility is increased to account for the improved mobilities in silicon-germanium (assumed to be 2500 cm$^2$/Vs for NMOS), and secondly, the capacitive effect of the oxide layer is modified to include the capacitive effect of the silicon cap layer. The new “oxide capacitance” is the capacitance which is obtained from the series combination of the oxide capacitance and the silicon cap layer capacitance:

\[
\frac{1}{C_{\text{ox,new}}} = \frac{1}{C_{\text{ox}}} + \frac{1}{C_{\text{si}}} \quad \text{(EQ. A94)}
\]

\[
C_{\text{ox,new}} = \frac{C_{\text{ox}}C_{\text{si}}}{C_{\text{ox}} + C_{\text{si}}} \quad \text{(EQ. A95)}
\]

From this it is possible to determine the new “oxide thickness” value to be used in the model:

\[
t_{\text{ox,new}} = \frac{\varepsilon_{\text{ox}}}{C_{\text{ox,new}}} = t_{\text{ox}} + \left(\frac{\varepsilon_{\text{ox}}}{\varepsilon_{\text{si}}}\right)t_{\text{si}} \quad \text{(EQ. A96)}
\]

There are other modifications which can be made to the model to incorporate other effects such as surface inversion; such work is left as an area of future research.
Appendix C: User’s Manual

The semiconductor device parameter space exploration tool can be used to provide a circuit or device designer with feasible parameter combinations for a given circuit. It takes as input a set of electrical criteria and displays a set of parameter combinations which are feasible for a given technology. The user can specify the electrical criteria, parameter ranges to explore, fixed parameter values, and device model to use with this tool. In addition, the user has the option of adding more models to the tool, saving generated data to disk, and formulating penalty functions which can be used with the optimization module.

This document serves as a user’s manual for using the device parameter space exploration tool. It includes procedures for:

* starting up the tool

* adding new models to the system

* setting fitting parameters for a given model

* semiconductor device parameter space generation

* plotting utility options
System Requirements

The tool is written in Java using JDK 1.02. The tool exists as both as Java applet and a Java application; the Java application allows additional features not included with the applet format, such the option for saving and loading data from the local file system. The requirement for running the applet is a Java-enabled Web browser. The requirement for running the application is a system which has the JDK-1.02 interpreter.

Start-up Procedure

For the Java applet form:

1. Start Java-compatible web browser.
2. Go to URL: http://www-mtl.mit.edu/~lee21/dset/test.html
3. Click on Button marker “Start DSET”. (Screen shot shown on next page)

For the application:

1. Go to directory with main Java class files (directory with “DSETApp.class”)
2. Enter the command: “java DSETApp”.

In both cases the graphical user interface for the main tool should appear. A screen shot of the tool appears on the next page.
Screen shot of applet start-up screen

Screen shot of graphical user interface
Generating Parameter Spaces

The user interacts with the tool and provides inputs via dialog boxes which appear when the user selects buttons on the main GUI window. Here is a sample procedure to generate a two-dimensional parameter space of substrate doping-oxide thickness combinations for substrate dopings in the range $10^{16}$ to $10^{17}$ and an electrical specification of drain current of 1 mA for $V_d = 1.5$ V, $V_g = 2$ V, $V_s = 0$V, $V_b = 0$ V. For this test case, assume that the layout parameters for the given transistor are desired to be $W=3$ μm, $L=0.5$ μm. Furthermore assume the technology to be explored is bulk CMOS, using the charge-sheet model.

Step 1: Specify the model.

Select the choice widget below the “Select Model:” label. Choose “Bulk Surf. Pot.(NMOS)” as the device model to be used.

Step 2: Specify Fixed Parameters.

Select the “Set Params” button on the main GUI. This brings up the fixed parameter value specifier dialog box.

Enter “3” into the field for “W” and “0.5” into the field for L. Leave all other parameters to their default values.

Step 3: Specify Swept and Calculated Parameters.

Select the “Set Vars” button on the main GUI. This brings up the parameter selections dialog box.

Select the choice widget next to the “Sweep” label. Scroll down the list of options to choose “Na”. Enter “1e16” in the “Start” field, “1e17” in the “Stop” field, and “1e15” in the “Step” field.
Select the choice widget next to the “Calculate” label. Scroll down the list of options to choose “tox”. Enter “1e-5” in the “Upper Bound:” field, and “5e-7” in the “Lower Bound:” field.

Select the “Okay” button to close the parameter selections dialog box.

Step 4: Select the Electrical Criteria.

Select the “Set Elec Crit.” button on the main GUI. This brings up the electrical specifications dialog box.

Select the units choice widget to the right of the “Drain Current:” field. Choose “mA” from the list of options. Enter “1” in the “Drain Current:” field.

Enter “1.5” in the “Drain Voltage:” field, “2” in the “Gate Voltage:” field, “0” in the “Source Voltage:” field, and “0” in the “Bulk Voltage:” field.

Select the “Okay” button to close the electrical specifications dialog box.

Step 5: Run the Engine.

Select the “Run Engine” button from the main GUI. Wait until the message “Finished calculations!” appears on the text window on the main GUI.

Step 6: View the output.

Select the “Autoscale” button to autoscale the generated data for viewing.

The output should look similar to the figure shown on the next page:
Sample Parameter Set Generation
Using the Plotting Utility

The plotting panel has several options which can be used to view X-Y data. There are 5 different widgets on the plotting utility.

**Set Color Choice Widget:**

This widget may be used to select the color of the NEXT plotted data.

**Set Bounds:**

This button spawns a dialog box where the user may input the x and y axis limits for the plot.

The user also has the option of holding down the middle mouse button (or the “Alt” key with the left mouse button) while dragging a rectangular area on the plot. The plotting utility will then zoom into the indicated area.

**Autoscale:**

This button autoscales the graph.

**Clear Data:**

This button deletes all data from the currently shown graph.

**Select Graph:**

This button spawns a choice widget box which the user can utilize to specify the particular graph to view.
Using the Optimizer

Here is a sample procedure for utilizing the optimization tool. This procedure can be used to generate the results from Test Scenario Case #4 in Chapter 6.

Step 1: Start the Optimizer.

Select the “Bulk Surf. Pot. (NMOS)” model from the model choice widget. Select the “Set Params” button from the main GUI and enter: “0.9” in the “W” field, “0.45” in the “L” field, “8e20” in the “ng” field, and “5e-7” in the “Xj” field. Select “Okay” to close the parameter selection dialog box.

Select the “Optimize” button from the main GUI. This brings up the optimization tool window, shown below:

![Optimization Tool Window]

Step 2: Select a model penalty function.

Select “MPF1” from the list in the upper left corner of the optimizer GUI. Once properly selected, the choice should be highlighted. “MPF1” is a generic model penalty function object, created when the optimization tool is first spawned, and having no characteristics other than the initialized values set for all instantiated model penalty function objects.
Step 3: Select Optimization Parameters.

Select the “Select Optimization Parameters” button. This brings up a list box.

Choose “Na” and “tox” from the options in the list box. Ensure that BOTH choices are highlighted.

Enter “1e-10” for tolerance in the “ftol” field.

Select the “Okay” button to close the list box.

Step 4: Select Initial Values.

Select the “Initial Values” button from the optimizer GUI. This brings up the initial values selection dialog box.

Enter “1e15” in the “Na” field and “1e-6” in the “tox” field.

Select the “Okay” button to close the dialog box.

Step 5: Select the Weights.

Select the “Set weights” button from the optimizer GUI. This brings up the weights specification dialog box.

Enter “1” in the “Na” field and “1” in the “tox” field. Enter “0” in the “Beta” field.

Select the “Okay” button to close the dialog box.

Step 6: Select the Electrical Criteria.

Select the “Add Elec. Crit” button from the optimizer GUI. This brings up the electrical specification dialog box.
Enter “1” in the “Lambda” field.

Select “\(\mu\text{A}\)” from the drain current units choice widget, then enter “330” in the “Drain current:” field.

Enter “3” in the “Drain Voltage:” field.

Enter “2” in the “Gate Voltage:” field.

Enter “0.5” in the “Source Voltage:” field.

Enter “0” in the “Bulk Voltage:” field.

Select the “Set” button. Do not select the “Okay” button yet.

Repeat once more, entering “216”, “1”, “2”, “0.5”, and “0”, and selecting “Set” when finished. Make sure that the drain current units widget is still set on “\(\mu\text{A}\)” and that the lambda value is still set to 1.

Repeat once more, entering “161”, “2”, “1.5”, “0.5”, and “0”, and selecting “Okay” when finished. This closes the electrical specification dialog box.

**Step 7: Select an Optimization Routine.**

Select the optimization routine choice widget in the lower left corner of the optimizer GUI. Choose “Simplex”.

**Step 8: Run the Optimization.**

Select the “Optimize” button. Wait for the optimization routine to complete and print the result in the text window. The result should match what is shown on the next page:
Optimizer Results Screen

END SIMPLEX
37.9535    -14.0396
37.9535    -14.0396
37.9535    -14.0396

Optimal point is:
[0] = 3.04093e+16
[1] = 7.99258e-07
Adding New Models to the Tool

The procedure for adding a new model object to the list of objects to be used with this tool is as follows:

1. Convert the model equations to Java code, insuring that the Java code follows the model object protocol specified in the DeviceModel interface.

2. Compile the Java code into a model object.

3. Move the model object into the models package.

4. Add the appropriate line(s) to the model file. (See template below)

Once the procedure has been followed, execution of the tool should lead to incorporation of the new model object. Note that recompilation of the system source code is NOT required in order to add new models to the system.

Model File Template

<table>
<thead>
<tr>
<th>DSET MODEL CLASS INFORMATION</th>
<th>DSET MODEL CLASS INFORMATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;MODEL CLASS NAME 1&gt;</td>
<td>BulkSurfPotModel</td>
</tr>
<tr>
<td>DEFAULTS</td>
<td>Na</td>
</tr>
<tr>
<td></td>
<td>1e17</td>
</tr>
<tr>
<td></td>
<td>tox</td>
</tr>
<tr>
<td></td>
<td>1e-7</td>
</tr>
<tr>
<td>END</td>
<td>SOIASModel</td>
</tr>
<tr>
<td></td>
<td>DEFAULTS</td>
</tr>
<tr>
<td></td>
<td>tox</td>
</tr>
<tr>
<td></td>
<td>1e-7</td>
</tr>
<tr>
<td></td>
<td>tbox</td>
</tr>
<tr>
<td></td>
<td>1e-8</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>

Template

Notes:

(1) Specification of default values of parameter is optional
(2) Delimiters between parameter name and value can be any form of whitespace (spaces or tabs).
The DeviceModel interface:

package models;

public interface DeviceModel {

    /* Constants defined here */
    public static final double K=1.38000e-23; /* J/K */
    public static final double Q=1.6000e-19; /* Coulomb */
    public static final double EOX=3.45150e-11; /* F/m */
    public static final double ESI=1.03545e-10; /* F/m */

    // Defines the electrical I-V relationship
    double calc_i(double volt[]);

    // Allows specification of a name
    String get_name();

    // Sets a parameter's real value, in global units
    void setparam_value(String name, double val);

    // Returns a parameter's value, in global units
    double get_param_value(String name);

    // Sets a parameter's value, in most commonly used units
    void set_param_cvalue(String name, double val);

    // Returns a parameter's value, in most commonly used units
    double get_param_cvalue(String name);

    // Returns a specified secondary parameter (e.g., threshold voltage)
    double get_sec_param(String name);

    // Returns a parameter's conversion factor from global units to most
    // common units
    double get_cf(String name);

    // Returns a parameter's most common units descriptor
    String get_unit_str(String name);

    // Returns the GenericParam object to the user
    GenericParam get_param(String name);

    // Returns true if the parameter descriptor describes
    // a parameter used in the model, false otherwise
    boolean param_exists(String name);

    // Returns an array of descriptor names for parameters
    // in the model.
    String[] get_param_names(String x);

    // Sets a parameter's default value
void set_default_value(String name, double val);

// Resets all parameters to their defaults
void set_defaults();

// Sets a parameter’s valid range
void set_bounds(String name, double lower_bound, double upper_bound);

// Retrieve a parameter’s valid range
double[] get_bounds(String name);

// Updates all variables within a model, including
// secondary variables.
void update_all();

// Prints parameters and value of a given object to standard output.
void print_all_params();

// Returns a copy of a model object.
public DeviceModel make_copy();

// Returns an optimization informational object. If extra
// information about the model equations is known, then
// such information can be placed within an optimization
// object here.
public OptimizationInfo optimization_info();

}
Sample Model Object code template:

The template shown in this section is can be used as an example for implementing new models for use with this system. The model equations should be added in the calc_i method (there are no equations in the template). This model assumes the existence of environmental, process, and layout parameters. Comments in the code provide directions for addition of parameter sets, and instructions for specific modifications.

```java
package models;
import java.lang.*;

public class MyModel implements DeviceModel{

    String name="My Model";

    // NOTE: THIS TEMPLATE ASSUMES PARAMETER GROUPINGS
    // OF ENVIRONMENTAL, PROCESS, AND LAYOUT
    // PARAMETERS. TO ADD ADDITIONAL PARAMETER GROUPS,
    // SIMPLY ADD A SIMILAR CODE LINE. THE CONSTRUCTOR
    // FOR THE PARAMSET CLASS TAKES AS INPUT THE MAXIMUM
    // NUMBER OF PARAMETERS WHICH CAN BE HELD IN A PARAMETER
    // GROUP.
    ParamSet env_params=new ParamSet(10);
    ParamSet proc_params=new ParamSet(20);
    ParamSet lay_params=new ParamSet(10);

    // Declare some variables to hold "secondary" parameter values
    double Cox, gamma, phi_t;

    /**
     * public MyModel(String id);
     * The constructor, which is used to specify the
     * type of transistor(e.g. N-type or P-type). If
     * called with no argument then it defaults to
     * NMOS.
     */
    public MyModel(){
        this("NMOS");
    }

    public MyModel(String id){
        if (id.equals("PMOS")){
            name=name+"(PMOS)";
        } else {
            name=name+"(NMOS)";
        }
    }
```
create_params();
set_defaults();
return;
}

/**
 * private void create_params();
 * This method initializes the parameter set
 * groups and adds the parameter descriptors,
 * initial values, and conversion factors.
 */
private void create_params(){

// ADD ALL PARAMETERS HERE
// FORMAT: NAME, VALUE (IN MKS UNITS), CONVERSION FACTOR,
// MOST COMMONLY USED UNITS’ DESCRIPTOR,
// LOWER BOUND AND UPPER BOUND ON PARAMETER
// VALUES (IN MKS UNITS)
// EXAMPLES SHOWN BELOW:

env_params.add_param("T",300.0,1.0,"K",0,400);
proc_params.add_param("Na",3.5e23,1e6,"cm^-3",1e21,1e25);
proc_params.add_param("tox",9e-9,1e-2,"cm",1e-10,1e-6);
lay_params.add_param("W",0.9e-6,1e-6,"um",0.1e-6,200e-6);
return;
}

/**
 * public void setDefault_value(String name, double val);
 * This method is used to set a specific parameter’s default
 * value.
 */
public void setDefault_value(String name,double val){

// NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
// IT SHOULD BE MODIFIED ONLY IF ANOTHER PARAMETER
// GROUPING (I.E. A PARAMSET CLASS CORRESPONDING TO
// FITTING PARAMETERS) IS ADDED TO THIS MODEL OBJECT.
if (env_params.param_exists(name)){
   env_params.set_param_defaultc(name,val);
}
else if (proc_params.param_exists(name)){
   proc_params.set_param_defaultc(name,val);
}
else if (lay_params.param_exists(name)){
   lay_params.set_param_defaultc(name,val);
}
}

/**
 * public void setDefaults();
 * This method is used to reset all parameters within the
 * model to their default values.
 */
public void setDefaults(){

}
env_params.reset_all();
proc_params.reset_all();
lay_params.reset_all();
update_all();

/**
 * public void update_all();
 * This method is used to update any internal variables
 * within the model object, including variables corresponding
 * to "secondary" parameters.
 */
public void update_all(){
// NOTE: IF ANY INTERNAL VARIABLES ARE ADDED, THEN
// ANY UPDATES TO THEM SHOULD BE PERFORMED HERE.
calc_sec_params();
}

/**
 * public String get_name();
 * This method returns a String which is a description of
 * the model. The name for the model should be set
 * when the constructor is called.
 */
public String get_name(){
// NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
    return name;
}

/**
 * public void set_param_value(String name, double val);
 * This method is used to set a specific parameter’s
 * value, in MKS units. To set a specific parameter’s
 * value in most commonly used units, the method
 * set_param_cvalue() should be invoked.
 * Variable values are most useful in equation evalutions
 * when all of the units are on some global scale (such
 * as MKS units), but are most useful to the user when
 * presented in units in which they are most commonly
 * expressed. The parameter value is thus stored in
 * MKS units, but can be recalled in terms of most
 * commonly used units via the get_param_cvalue()
 * method.
 */
public void set_param_value(String name, double val){
// NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
// IT SHOULD BE MODIFIED ONLY IF ANOTHER PARAMETER
// GROUPING (I.E. A PARAMSET CLASS CORRESPONDING TO
// FITTING PARAMETERS) IS ADDED TO THIS MODEL OBJECT.
if (env_params.param_exists(name)){
    env_params.set_param(name,val);
} else if (proc_params.param_exists(name)){
    proc_params.set_param(name,val);
} else if (lay_params.param_exists(name)){
    lay_params.set_param(name,val);
}
update_all();

/**
 * public double get_param_value(String name);
 * This method is used to retrieve a parameter’s value, in
 * MKS units.
 */
public double get_param_value(String name){
    // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
    return (get_param(name)).get_value();
}

/**
 * public void set_bounds(String name, double lower_bound,
 * double upper_bound);
 * This method is used to set a constraint of the range of parameter
 * values which a specific parameter can be. The values of the lower
 * and upper bounds should be given in terms of most commonly used
 * units.
 * If a parameter is requested to be set to a particular value (via
 * the set_param_value() or set_param_cvalue() method) then the value
 * that the parameter will be set to is guaranteed to be within the
 * specified bounds. If the original specified value is not within
 * the bounds, then the value of the parameter is set to the nearest
 * specified bound.
 */
public void set_bounds(String name, double lower_bound, double upper_bound){
    // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
    // IT SHOULD BE MODIFIED ONLY IF ANOTHER PARAMETER
    // GROUPING (I.E. A PARAMSET CLASS CORRESPONDING TO
    // FITTING PARAMETERS) IS ADDED TO THIS MODEL OBJECT.
    if (env_params.param_exists(name)){
        (env_params.get_param(name)).set_bounds(lower_bound,upper_bound);
    } else if (proc_params.param_exists(name)){
        (proc_params.get_param(name)).set_bounds(lower_bound,upper_bound);
    } else if (lay_params.param_exists(name)){
        (lay_params.get_param(name)).set_bounds(lower_bound,upper_bound);
    }
/**
 * public double[] get_bounds(String name);
 * This method is used to retrieve the constraint information
 * about a specific parameter. The lower bound is returned
 * as the first element in the array, and the upper bound is
 * returned as the second element.
 */
public double[] get_bounds(String name){
    // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
    if (param_exists(name)){
        return (get_param(name)).get_bounds();
    }
    return null;
}

/**
 * public GenericParam get_param(String name);
 * This method is used to retrieve the actual parameter object
 * (an instantiation of the GenericParam class) with the
 * specified name. Specific GenericParam methods can then
 * be invoked on the retrieved parameter object.
 */
public GenericParam get_param(String name){
    // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
    // IT SHOULD BE MODIFIED ONLY IF ANOTHER PARAMETER
    // GROUPING (I.E. A PARAMSET CLASS CORRESPONDING TO
    // FITTING PARAMETERS) IS ADDED TO THIS MODEL OBJECT.
    if (env_params.param_exists(name)){
        return env_params.get_param(name);
    }
    else if (proc_params.param_exists(name)){
        return proc_params.get_param(name);
    }
    else if (lay_params.param_exists(name)){
        return lay_params.get_param(name);
    }
    else return (new GenericParam("NULLPARAM",0.0));
}

/**
 * public void set_param_cvalue(String name, double val);
 * This method is used to set a parameter’s value in terms
 * of its most commonly used units.
 */
public void set_param_cvalue(String name, double val){
    // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
    set_param_value(name, val*get_cf(name));
}
/**
public double get_param_cvalue(String name);
* This method is used to retrieve a specific
* parameter’s value, in terms of most commonly
* used units.
*/

public double get_param_cvalue(String name)
    { // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
        return get_param_value(name)/get_cf(name);
    }
/**
* public double get_cf(String name);
* This method is used to retrieve the conversion
* factor (from MKS units to most commonly used units)
* of a specific parameter.
*/

public double get_cf(String name)
    { // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
        return (get_param(name)).get_cf();
    }
/**
* public String get_unit_str(String name);
* This method is used to retrieve the units
* descriptor for a parameter’s most commonly used
* units.
*/

public String get_unit_str(String name)
    { // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
        return (get_param(name)).get_unit_str();
    }
/**
* public boolean param_exists(String name);
* This method is used to query whether a
* specific parameter exists within a given
* model object.
*/

public boolean param_exists(String name)
    { // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
        // IT SHOULD BE MODIFIED ONLY IF ANOTHER PARAMETER
        // GROUPING (I.E. A PARAMSET CLASS CORRESPONDING TO
        // FITTING PARAMETERS) IS ADDED TO THIS MODEL OBJECT.
        return (env_params.param_exists(name) ||
                    proc_params.param_exists(name) ||
                    lay_params.param_exists(name));
    }
/**
* public void print_all_params();
* This method is used to display the values of
* all of the parameters within the model object.
*/
public void print_all_params()
{
    // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
    // IT SHOULD BE MODIFIED ONLY IF ANOTHER PARAMETER
    // GROUPING (I.E. A PARAMSET CLASS CORRESPONDING TO
    // FITTING PARAMETERS) IS ADDED TO THIS MODEL OBJECT.
    int num;
    num=env_params.get_num_params();
    for (int i=0;i<num;i++)
    {
        output_message(env_params.get_param(i).get_name() + " is "+
                        Double.toString(env_params.get_param(i).get_cvalue()) + " +
                        env_params.get_param(i).get_unit_str());
    }
    num=proc_params.get_num_params();
    for (int i=0;i<num;i++)
    {
        output_message(proc_params.get_param(i).get_name() + " is "+
                        Double.toString(proc_params.get_param(i).get_cvalue()) + " +
                        proc_params.get_param(i).get_unit_str());
    }
    num=lay_params.get_num_params();
    for (int i=0;i<num;i++)
    {
        output_message(lay_params.get_param(i).get_name() + " is "+
                        lay_params.get_param(i).get_cvalue()) + " +
                        lay_params.get_param(i).get_unit_str());
    }
}

/**
* public String[] get_param_names(String x);
* This method is used to retrieve a list of parameter
* names from a specific parameter group, or from the
* entire set of parameters within a model object.
* This string input should be a descriptor which
* specifies the parameter group (e.g. “process” or “all”).
*/
public String[] get_param_names(String x)
{
    // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
    // IT SHOULD BE MODIFIED ONLY IF ANOTHER PARAMETER
    // GROUPING (I.E. A PARAMSET CLASS CORRESPONDING TO
    // FITTING PARAMETERS) IS ADDED TO THIS MODEL OBJECT.
    if (x.equalsIgnoreCase("env") || x.equalsIgnoreCase("environment")){
        return env_params.get_param_names();
    }
    else if (x.equalsIgnoreCase("proc") || x.equalsIgnoreCase("process"){
        return proc_params.get_param_names();
    }
    else if (x.equalsIgnoreCase("lay") || x.equalsIgnoreCase("layout"){
        return lay_params.get_param_names();
    }
    else if (x.equalsIgnoreCase("all") || x.equalsIgnoreCase("every"){
        return get_param_names();
    }
    else return null;
}
public String[] get_param_names();

This method is used to retrieve the entire set of parameter descriptors from a model object.
This method is invoked from the method
public String[] get_param_names(String x)
when the input is “all” or “every”.
*
public String[] get_param_names(){
    // NOTE: THIS CODE BLOCK MAY BE USED FOR ANY MODEL.
    // IT SHOULD BE MODIFIED ONLY IF ANOTHER PARAMETER
    // GROUPING (I.E. A PARAMSET CLASS CORRESPONDING TO
    // FITTING PARAMETERS) IS ADDED TO THIS MODEL OBJECT.
    int size=env_params.getnum_params() +
               proc_params.get_num_params() +
               lay_params.get_num_params();
    String rarr[]=new String[size];
    int index=0;
    String namearr[];
    namearr=env_params.get_param_names();
    for (int i=0;i<env_params.get_num_params();i++,index++){
        rarr[index]=namearr[i];
    }
    namearr=proc_params.get_param_names();
    for (int i=0;i<proc_params.get_num_params();i++,index++){
        rarr[index]=namearr[i];
    }
    namearr=lay_params.get_param_names();
    for (int i=0;i<lay_params.get_num_params();i++,index++){
        rarr[index]=namearr[i];
    }
    return rarr;
}

/**
 * public DeviceModel make_copy();
 * This method is used to clone a copy of a model object.
 */
public DeviceModel make_copy(){
    // NOTE: TO MODIFY THIS METHOD FOR A NEW MODEL
    // CHANGE THE INSTANTIATION OF THE DEVICEMODEL
    // VARIABLE TO THE NAME OF THE NEW MODEL OBJECT.
    // ALL OTHER CODE LINES CAN REMAIN THE SAME.
    DeviceModel dm;
    if (type==PMOS) dm=new MyModel("PMOS");
    else dm=new MyModel("NMOS");
}
String param_names[]=get_param_names("all");

for (int i=0;i<param_names.length;i++)
    dm.set_param_value(param_names[i],
    this.get_param_value(param_names[i]));

return dm;
}

/**
 * public OptimizationInfo optimization_info();
 * This method returns an object which contains
 * information about a model’s equations. Such
 * information may be used by an optimization
 * program for selection of a particular routine
 * or faster completion of an algorithm.
 */
public OptimizationInfo optimization_info(){
    // NOTE: ADDITION OF OPTIMIZATION INFORMATION
    // ABOUT A MODEL OBJECT’S EQUATIONS CAN BE
    // ADDED USING THE “ADD_PROPERTY” METHOD.
    OptimizationInfo OPI=new OptimizationInfo(this);
    OPI.add_property("monotonic_all");
    return OPI;
}

/**
 * public void calc_sec_params();
 * This method is used to calculate secondary parameters
 * which are used in model equations. For examples, the
 * body coefficient gamma is calculated from substrate doping
 * and oxide thickness.
 */
public void calc_sec_params(){
    // NOTE: VALUES OF PARAMETERS WITHIN THE PARAMETER
    // SETS SHOULD BE RETRIEVED USING THE GET_PARAM_VALUE
    // METHOD WHICH RETURNS A PARAMETER’S VALUE IN TERMS
    // OF GLOBAL (MKS) UNITS. IN THIS WAY, ALL RETRIEVED
    // PARAMETER VALUES ARE ON THE SAME SET OF UNITS.

    // RETRIEVE PARAMETER VALUES
    double tox=get_param_value("tox");
    double Na=get_param_value("Na");
    double T=get_param_value("T");

    // CALCULATE SECONDARY PARAMETERS
    Cox=EOX/tox;
    gamma=Math.sqrt(2*ESI*Q*Na)/Cox;
    phi_t=K*T/Q;
}
/**
 * public double get_sec_param(String name);
 * This method is used to retrieve secondary
 * parameter values.
 */
public double get_sec_param(String name){
    // NOTE: ADDITION OF EXTRA SECONDARY PARAMETERS
    // SHOULD BE ACHIEVED BY ADDING AN EXTRA BRANCH
    // TO THE IF-ELSE BRANCH BELOW.
    update_all();
    if (name.equals("gamma")) return gamma;
    else if (name.equals("phi_t")) return phi_t;
    else if (name.equals("Cox")) return Cox;
    else return 0.0;
}

/**
 * public double calc_i(double volts[]);
 * This method is used to calculate the output
 * current as a function of the input voltages.
 * All parameter values should be extracted from
 * the parameter sets. Secondary parameter values
 * should be updated using update_all() to insure
 * valid values.
 * This method should contain the model equations
 * which are used to calculate the desired quantity
 * (e.g., drain current for specified voltage
 * values)
 */
public double calc_i(double volts[]){
    // Invoke update_all to update
    // secondary variables.
    update_all();

    // Declare a variable to hold return value.
    double i;

    // Extract voltage values from the
    // input voltage array.
    double drain_v=volts[0];
    double gate_v=volts[1];
    double source_v=volts[2];
    double bulk_v=volts[3];
    double Vds=drain_v-source_v;
    double Vbs=bulk_v-source_v;
    double Vgb=gate_v-bulk_v;
    double Vgs=gate_v-source_v;

    // Retrieve all parameter values from
    // parameter groups. Use the method
    // get_param_value since values should
// all be expressed in the same scale.
// NOTE: ANY EXTRA PARAMETERS ADDED TO MODEL
// SHOULD BE RETRIEVED FOR USE IN MODEL
// EQUATIONS.
double T=get_param_value("T");
double Na=get_param_value("Na");
double tox=get_param_value("tox");
double W=get_param_value("W");

if (name.equals("MyModel(NMOS)")){
    i=0.0;
    // INSERT N-TYPE MODEL EQUATIONS HERE
}
else {
    i=0.0;
    // INSERT P-TYPE MODEL EQUATIONS HERE
}

// RETURN VALUE HERE
return i;
}