**Finding All DC Operating Points Using
Interval-Arithmetic-Based Verification Algorithms**

by

Itrat Ahmed Akhter

B.A., Colby College, 2016

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF SCIENCE

(Computer Science)

The University of British Columbia

(Vancouver)

November 2018

© Itrat Ahmed Akhter, 2018

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Finding All DC Operating Points Using Interval-Arithmetic-Based Verification Algorithms

submitted by **Itrat Ahmed Akhter** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Science**.

Examining Committee:

Mark R. Greenstreet, Computer Science
Supervisor

Alan J. Hu, Computer Science
Co-supervisor

Abstract

Finding operating points of circuits is a crucial first step for simulation and verification. Traditional operating point analyses such as homotopy analysis do find an operating point. However, a circuit may have many DC equilibria, and failing to consider an unintended initial condition can lead to failures escaping to the physical silicon. This work presents a method for finding all DC equilibrium points; this approach is based on interval-arithmetic-based verification algorithms; and our open-source implementation supports state-of-the-art short-channel device models. This work also presents what we believe to be the first, completely automatic verification of the Rambus ring-oscillator start-up problem. Our method offers large performance and scalability advantages when compared with the dReal and Z3 SMT solvers.

Lay Summary

DC operating point analysis is a crucial step in simulation, analysis, and verification of analog circuits. Finding all DC operating points prevents errors in designs from escaping to actual chips. This research work presents a novel method of finding all DC operating points based on an interval-arithmetic-based verification called the Krawczyk method. This method is more scalable than current state-of-the-art SMT solvers like Z3 and dReal. It also allowed the first completely automatic verification of a circuit that was presented as a challenge problem by researchers at Rambus.

Preface

Parts of this work has been accepted for publication in *The Proceedings of the 2019 Design Automation and Test in Europe Conference* (DATE'2019) as a paper titled “Finding All DC Operating Points Using Interval Arithmetic Based Verification Algorithms” where I am the primary author. The key ideas from this thesis are presented in the paper. Parts of Chapter 1 and Chapter 2 have been adopted from this paper. I am responsible for the rest of the chapters in this thesis. In this work, I am responsible for all of the experiments and data analysis. I wrote the source code for this research and acknowledge the value of design discussions and debugging help from my supervisor, Mark Greenstreet. The transistor models build on code that was originally written by Mark Greenstreet and/or Justin Reiher – the co-authors of the DATE paper described above. The source code is located in <https://github.com/iaakhter/nonLinearSolver>, where each file and function is labeled with its author. Figure 5.3 and Figure 5.4 were inspired from [15] and [36] respectively with permission from Professor Mark Greenstreet who is one of the authors of both the papers.

Table of Contents

Abstract	iii
Lay Summary	iv
Preface	v
Table of Contents	vi
List of Tables	viii
List of Figures	x
Acknowledgments	xi
1 Introduction	1
2 Related Work	3
3 Background	7
3.1 Newton's Method	8
3.2 The Krawczyk Method	8
3.3 Linear Programming	9
4 Implementation	13
4.1 The Krawczyk Method and Bisection	13
4.2 Linear Programming as a Decision Procedure	20
4.3 Interval Arithmetic Implementation Details	24

4.4	Circuit Models	26
4.4.1	tanh Inverter Model	27
4.4.2	Long Channel MOSFET Transistor Model	28
4.4.3	Short Channel MOSFET Transistor Model	29
4.5	Possible Extensions	30
5	Results	31
5.1	Simple Circuits	33
5.2	Rambus Ring Oscillator	34
5.3	Schmitt Trigger	40
6	Conclusions	43
	Bibliography	45
A	Appendix	50
A.1	C++ Short Channel MOSFET MODEL Code	50
A.2	Informal Proof of the Number of DC Operating Points in the Ram- bus Ring Oscillator with Strong Cross-coupled Inverters	53

List of Tables

Table 4.1	Variation of Run-time with Variations in Repetitions of the Krawczyk Step in Single Run for a 4-stage Rambus Oscillator. The <code>modelType</code> indicates which model of inverters was used in the Rambus oscillator. <code>tanh</code> means that the inverter was modeled by a tanh function, and <code>LCMOSFET</code> means that the inverter was modeled by 2 CMOS transistors with a long channel MOSFET model. <code>gcc</code> indicates the ratio of strength of cross-coupled inverters to that of forward inverters in the Rambus ring oscillator. <code>kα</code> is the proportion of volume reduction threshold that defines how much the Krawczyk step should refine the volume of a hyperrectangle before stopping. <code>#bisection</code> indicates the number of bisections used in our method to solve the problem instance. The total run-time is indicated in the <code>Total</code> column.	17
-----------	--	----

Table 4.2	Variation of Run-time with Variation of Epsilon Inflation for a 4-stage Rambus Oscillator. The <code>modelType</code> indicates which model of inverters was used in the Rambus oscillator. <code>tanh</code> means that the inverter was modeled by a tanh function, and <code>LCMOSFET</code> means that the inverter was modeled by 2 CMOS transistors with a long channel MOSFET model. <code>gcc</code> indicates the ratio of strength of cross-coupled inverters to that of forward inverters in the Rambus ring oscillator. <code>epsilon (%)</code> indicates the proportion of hyper-rectangle interval length used for epsilon inflation. <code>#bisect</code> indicates the number of bisections used in our method to solve the problem instance. The total run-time is indicated in the <code>Total</code> column.	19
Table 4.3	Comparison of Bisection Types for a 4-stage Rambus oscillator. <code>tanh</code> means that the inverter was modeled by a tanh function, <code>LCMOSFET</code> means that the inverter was modeled by 2 CMOS transistors with a long channel MOSFET model, and <code>SCMOSFET</code> means that the inverter was modeled by 2 CMOS transistors with a short channel MOSFET model. <code>gcc</code> indicates the ratio of strength of cross-coupled inverters to that of forward inverters in the Rambus ring oscillator. <code>bisectType</code> indicates the type of bisection used in our method. <code>#bisect</code> indicates the number of bisections used in our method to solve the problem instance. The total run-time is indicated in the <code>Total</code> column.	21
Table 5.1	Inverter – comparison of our algorithm and dReal and Z3 . . .	33
Table 5.2	Inverter Loop – comparison of our algorithm and dReal and Z3 . . .	35
Table 5.3	Rambus Ring Oscillator – comparison of our algorithm and dReal . . .	37
Table 5.4	Rambus Ring Oscillator – runtime detail. Linear programming for short channel MOSFET model was not implemented and therefore the short channel MOSFET model results are not included.	38
Table 5.5	Schmitt Trigger – comparison of our algorithm and dReal . . .	41

List of Figures

Figure 3.1	Linear Constraints for $y = \tanh(-5 * x)$ for $-1 \leq x \leq 0$	10
Figure 3.2	Moved Constraint in Union of Linear Programs.	12
Figure 4.1	Bisection Procedures	20
Figure 5.1	Approximation of $\tanh(a * x)$ where $a = -5$ with a piece-wise polynomial function	32
Figure 5.2	Inverter Loops	34
Figure 5.3	Rambus Ring Oscillator	35
Figure 5.4	Schmitt Trigger.	39
Figure 5.5	DC Operating Points of Schmitt Trigger using the short channel MOSFET model	40
Figure 5.6	Eigenvalues of the Schmitt Triggers Unstable DC Operating Point	42

Acknowledgments

I would like to thank my supervisor Mark Greenstreet and my co-supervisor Alan Hu for guiding me through my research. I would also like to thank Sam Bayless who had a hand in helping me focus my research topic.

Last but not the least, I would not have had this wonderful opportunity to study at UBC without the support from my family and friends. My endless gratitude goes especially to my grandfather Md. Shamsuzzoha who taught me the value of education.

This work was supported in part by grants from Intel and NSERC.

Chapter 1

Introduction

DC operating point analysis is a crucial first step for simulation, analysis, and verification of analog and mixed-signal circuits. As described in [20], most analog circuits are intended to have linear transfer functions when analyzed in the appropriate domains. This linear behaviour is exhibited when the circuit is at its intended operating point. Showing that the circuit always converges to its intended operating point must account for the non-linear, large-signal properties of the circuit. Modern designs can make frequent mode transitions where digital control signals alter the operating points of analog circuits to optimize power and performance. If a circuit has more than one DC operating point, then failing to consider all possible operating points can lead to design errors escaping to actual chips (e.g. see [17]). Thus, there is an interest in finding all or many of the DC operating points.

As in most work on circuit simulation and verification, I model circuits using non-linear, ordinary differential equations. If v is the vector of node voltages, then the circuit defines a derivative function such that $\frac{d}{dt}v = f(v)$, and a DC operating point is a voltage vector, v_0 , such that $f(v_0) = 0$. There are two main approaches to finding DC operating points: homotopy-based approaches [14, 22, 24, 35] and SAT and SMT based methods [8, 10, 11, 18, 25, 34].

I present a novel approach of finding DC operating points. This approach is based on interval arithmetic and interval verification algorithms [27, 31]. Specifically I use the Krawczyk method [21] (see also [31, Sec. 13]) that can prove the existence and uniqueness of solution to a system of non-linear equations given in-

terval bounds.

By using interval verification algorithms, we can establish the existence of a unique solution to a system of non-linear equations in a hyper-rectangle. This has the advantage that in addition to finding a small hyper-rectangle that provides tight bounds for a given solution, we can also find a maximal hyper-rectangle that contains that solution and no other. This provides a simple mechanism for enumerating all solutions. The Krawczyk method also provides faster convergence than bisection alone, allowing our approach to scale to problems with hundreds of solutions.

The contributions of this work are as follows:

- I apply an interval verification algorithm [31] called the Krawczyk method to the verification of analog and mixed signal circuits. While other researchers have used interval arithmetic, we believe that this work is the first demonstration of these verification algorithms for analog circuit verification.
- I present the first, completely automatic verification of the Rambus Ring Oscillator start-up challenge problem. Prior work includes a proof that relies on manual proofs of monotonicity properties [15], adding assumptions that DC equilibria in certain regions must be unstable [34], or using particle filtering methods that provide a visualization, but not a proof [33].
- To provide a more complete comparison with existing SAT and SMT based tools (Z3 and dReal), I also analyze some simpler circuits including an inverter, inverter loops involving up to 4 inverters, and a Schmitt trigger. I also show how this approach allows us to compare the metastability resolution property of the Schmitt trigger with that of a two-inverter loop.

In this work, I have focused on DC operating point analysis because it is an area of practical importance, example problems are readily available, and we could compare our results with other methods. The problem of finding solutions to systems of non-linear equations or refuting the existence of such solutions is of interest in many fields including non-linear circuit design, robotics, and control systems, and as an interesting problem of its own right in pure and applied mathematics. In this dissertation, I focus on the DC-operating point problem as a source of motivating examples.

Chapter 2

Related Work

SPICE circuit simulation can be used to find DC operating points [28]. SPICE uses the Newton-Raphson method to find an operating point. However, convergence is not always guaranteed. It is also difficult to find more than one operating point unless some knowledge about the location of such points already exists. In SPICE, additional steps involving homotopy methods are used if the Newton-Raphson step fails to converge.

Homotopy methods derive a variation of the circuit for which finding the operating point is straightforward. A homotopy function is used to interpolate between the simple model and the actual circuit. Under appropriate conditions, the DC operating point(s) will move smoothly as the interpolation parameter is varied. For example, a typical homotopy function to solve $f(x) = 0, f : R^n \rightarrow R^n$ is $H(x, \lambda) = \lambda f(x) + (\lambda - 1)(g(x))$. This transforms the isolated solution points in R^n to a set of solution of curves in $R^n \times [0, 1]$. The starting simple solution is at $g(x) = 0$. As we vary λ from 0 to 1, we trace a curve from the starting solution to the actual solution. The efficiency of tracing depends on the characteristics of the curve - if they are long or ill-conditioned. λ can be chosen to control a specific circuit component - for example, voltage source or a resistor. G-min stepping and source stepping are some of the classic continuation methods used in many SPICE versions. This basic form of homotopy usually leads to one single solution. However, problems may arise in reaching a solution at $\lambda = 1$, if for example there is a bifurcation on the path leading to the $\lambda = 1$ point or if the path diverges to infinity

as λ approaches 1. Melville *et al.* tackle these problems by applying a class of homotopies known as “probability-one” homotopies to the simulation of non-linear circuits [24]. These homotopies have properties that guarantee that a solution can be reached at $\lambda = 1$.

There are also ways to find multiple solutions using homotopy methods. The two main ways are the multi-start approach and the extended curve approach [22]. In the multi-start approach, a number of different starting points are used at $\lambda = 0$. The goal is to follow their continuation paths to reach different solutions at $\lambda = 1$. However, with this approach the same solution could be reached with these different continuation paths and finding all solutions is not guaranteed. With the extended curve-following approach, a single solution path is followed beyond $\lambda = 1$ with the hope that the curve will reverse direction and pass through $\lambda = 1$ multiple times thus finding multiple solutions. Just like the multi-start approach, finding all solutions is not guaranteed if the path does not pass through multiple solutions.

Wolf and Sandars use a multi-parameter version of extended curve homotopy methods to find multiple solutions [35]. In a multiple parameter homotopy, a homotopy function has more than one interpolation parameter. Lee and Chiang use a combination of the multi-start approach and extended curve approach. They formalize conditions under which their method is guaranteed to find a single homotopy path tracing all the solutions [22]. For circuits where it is not possible to find such a single path, they provide a systematic way of finding the starting points of each homotopy path. Tracing each homotopy path from the starting points leads to all the solutions. However, the number of homotopy paths might significantly affect the performance. Goldgeisser and Green discuss conditions by which the likelihood of finding multiple points by extended curve are increased [14]. They develop an algorithm based on a SPICE-like simulator to take advantage of such conditions to find multiple DC operating points. These papers present results for circuits with up to a few dozen DC operating points. In contrast, Chapter 5 presents results showing that our method scales to circuits with hundreds of operating points.

SAT and SMT based approaches can also be used to find multiple DC operating points. Tiwari *et al.* divide the range of voltage values for each node in a circuit into a moderate number of intervals and compute relations that must hold on these

quantized voltages at a DC operating point [34]. They use these relations as constraints that are fed into an SMT solver to solve the problem. Miller and Brewer do something similar, but instead of using an SMT solver they implemented their own combination of an Integer Linear Programming Solver and a SAT solver for their circuit analysis examples [25]. The run-time for such methods grows rapidly as finer quantizations are used, and these methods can fail to capture key properties of an analog circuit. For example, a key property of a DC operating point is whether or not it is stable. This is a simple check of the eigenvalues of the Jacobian of f at the operating point, but such properties are not amenable to SAT based formulations. For example, Tiwari *et al.* attempt to discard unstable operating points in their search by adding an unverified assumption that an unstable equilibrium point will have one node with a voltage below $0.2V_{dd}$ and either of the other two nodes with a voltage above $0.8V_{dd}$ [34]. Furthermore, these methods cannot guarantee the existence or uniqueness of the solutions found.

More recently, SMT solvers promise extension of SAT solvers to problems involving real arithmetic. For example, Z3 [8, 18] uses Collin’s cylindrical algebraic decomposition (CAD) algorithm [7] as a decision procedure for systems of polynomial equalities and inequalities. The projection step in the CAD algorithm involves the production of many new polynomials from the initial polynomial. The number of these polynomials grow super-exponentially with the number of variables which prevents CAD from being scalable. Jovanovic and Moura try to mitigate this problem by only applying CAD when a conflict is encountered [18]. CAD is used to find the conflicting core — the constraints responsible for the conflict. Chapter 5 presents results where Z3 is unable to solve circuit examples involving piecewise polynomial functions whereas our method is able to.

Other researchers have used interval arithmetic in an SMT context. In interval constraint propagation techniques, a *branch and prune* method combines interval arithmetic and constraint propagation. They use interval extensions of functions to prune out intervals and branch on intervals that cannot be pruned. Intervals not pruned can also be reduced by constraint propagation [5]. Franzle *et al.* solve Boolean combinations of non-linear constraints that can include transcendental functions by introducing a method that tightly integrates SAT solving methods with interval constraint propagation [10]. Their solver, iSAT, can only *refute* the exis-

tence of solutions in a given hyper-rectangle. To understand why this is the case, consider trying to solve for $f(x) = 0$ where x and $f(x)$ are vector valued. iSAT can find a small hyper-rectangle for x for which the bounds for each component of $f(x)$ are all small intervals that contain 0, but this does not guarantee that there is an x for which all components of $f(x)$ are exactly 0.

Gao *et al.* introduce their tool dReal, which is also based on interval constraint propagation, that can solve various kinds of non-linear functions by incorporating δ -decidable decision procedures [11]. A formula in dReal can be thought of as a collection of real-equalities and real-inequalities joined with boolean connectives. Each such equality or inequality is an atom. Let $\delta > 0$ be a user-specified tolerance for resolving these atoms. The formula is δ -satisfiable if there are interval bounds for each real-valued variable and boolean values for each atom, such that each equality or inequality is satisfied (or not) according to the value for the atom, for all values of the real-variables in their intervals *if the (in)equality is perturbed by at most δ* . Gao *et al.* have shown that a very wide range of problems arising in formal reasoning with real-valued models are δ -decidable [12].

Zaki *et al.* reported on using HySat (a solver with iSat as the core algorithm) for DC operating point problems [36] and found that they could only solve very simple examples with only a few operating points. These approaches are only able to produce intervals that may contain solution whereas our approach can guarantee existence and uniqueness of solutions in a hyper-rectangle.

This work is based on interval arithmetic and interval verification algorithms [27, 31]. Interval Newton method, interval Gauss Seidel and the Krawczyk method are some of the interval verification algorithms that can be used to prove the existence and uniqueness of solution to a system of non-linear equations given interval bounds. I specifically use the Krawczyk method [21] (see also [31, Sec. 13]). Our bisection-based search is similar to that of iSAT [9, 10] or dReal [11]. However, a direct implementation of interval arithmetic such as in iSAT can only refute the existence of solutions, it cannot prove their existence. Our approach can find all DC operating points of circuits in the form of hyper-rectangles each of which is guaranteed to contain a unique solution.

Chapter 3

Background

The main component of our approach is an interval-arithmetic-based verification algorithm called the Krawczyk method. As an optimization, I have also incorporated linear programming in our approach. This chapter provides a brief overview of these methods and related background.

The interval-arithmetic notation this thesis uses is based on [30]. $\mathbb{IR}$ denotes the set of intervals over the reals, $\mathbb{IR}^n$ denotes the set of n -dimensional hyper-rectangles, and $\mathbb{IR}^{n \times n}$ denotes the set of $n \times n$ interval matrices. $\text{mid}(\mathbf{X})$ denotes the midpoint of hyper-rectangle $\mathbf{X}$. Note that throughout this thesis, bold font variable names such as $\mathbf{X}$ denote interval-valued scalars, vectors, matrices, or functions, and non-bold names such as x denote non-interval values. For legibility and consistency with the interval-arithmetic literature, capital letters such as $\mathbf{X}$ denote hyper-rectangles, and lower case letters such as x denote points.

Let $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be a differentiable function. Then $F : \mathbb{IR}^n \rightarrow \mathbb{IR}^m$ denotes the *interval extension* of f where each arithmetic operation and elementary function is replaced with its interval arithmetic equivalent with outward rounding (see Chapter 4, Section 4.3 for more details about outward rounding). It is straightforward to show (see [30, Eq. 5.17]) that

$$x \in \mathbf{X} \Rightarrow f(x) \in F(\mathbf{X}) \quad (3.1)$$

In the case where f is known to be monotonic in each of its arguments (as is the

case for all of our circuit models), tighter interval bounds for $F(\mathbf{X})$ can be obtained by evaluating f at the corner of $\mathbf{X}$ that minimizes f and at the corner that maximizes f .

3.1 Newton's Method

Newton's method is a widely used numerical approach for finding an approximation for x such that $f(x) = 0$. The basic idea is to start with a guess, $x^{(0)}$, evaluate f and its derivative, f' , at $x^{(0)}$, and use a linear approximation of f based on its derivative to obtain an "improved" guess, $x^{(1)} = x^{(0)} - f(x)/f'(x)$. In the case that $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$, $J_f(x) \in \mathbb{R}^{n \times n}$ denotes the Jacobian of f at x :

$$J_f(x)_{i,j} = \frac{\partial f(x)_i}{\partial x_j} \quad (3.2)$$

where $f(x)_i$ denotes the i^{th} component of $f(x)$ and x_j denotes the j^{th} component of x . Starting from an initial guess, $x^{(0)}$, Newton's method computes a sequence of approximations, $x^{(1)}, x^{(2)}, \dots$ with

$$x^{(k+1)} = x^{(k)} - J_f(x^{(k)})^{-1} f(x^{(k)}), \quad k \geq 0 \quad (3.3)$$

3.2 The Krawczyk Method

The Jacobian operator can be extended to intervals with

$$J_f(\mathbf{X}) = \{J_f(x) \mid x \in \mathbf{X}\} \quad (3.4)$$

Typically, computing this set is impractical; so we compute an inclusion, $\mathbf{J}_f(\mathbf{X}) \supseteq J_f(\mathbf{X})$, using interval arithmetic methods. If f is convex (or anti-convex), then J_f is monotonic, and we can obtain tighter bounds for $J_f(\mathbf{X})$ by evaluating J_f at the relevant corners of $\mathbf{X}$. When extended using the interval Jacobian operator, Newton's method can be used to prove the existence of a solution to $f(x) = 0$ with $x \in \mathbf{X}$ (see [3]). This involves computing

$$\{\mathbf{J}_f(\mathbf{X}^k)^{-1} f(\text{mid}(\mathbf{X}^k))\}$$

which is generally computationally expensive or incurs a large over-approximation by computing $\mathbf{J}_f(\mathbf{X}^k)^{-1}$ using interval arithmetic. Furthermore, the method fails if $J_f(x)$ is singular for *any* $x \in \mathbf{X}$.

The Krawczyk method [21] avoids the most egregious problems of Newton's method by avoiding the computation of the inverse of the interval Jacobian matrix. Like the Newton or the interval Newton methods, the Krawczyk method is an iterative process. Let $\mathbf{X}^{(k)}$ be the interval bounds after the k^{th} iteration. Then the basic idea is that given an interval at iteration k , $\mathbf{X}^{(k)}$, we can define the Krawczyk operator -

$$\begin{aligned} K(\mathbf{X}^{(k)}) = & \text{mid}(\mathbf{X}^{(k)}) - Cf(\text{mid}(\mathbf{X}^{(k)})) \\ & + (I - C\mathbf{J}_f(\mathbf{X}^{(k)}))(\mathbf{X}^{(k)} - \text{mid}(\mathbf{X}^{(k)})) \end{aligned} \quad (3.5)$$

where K denotes the Krawczyk operator; C is a non-singular preconditioning matrix typically chosen to be the inverse of $f'(\text{mid}(\mathbf{X}^{(k)}))$; I is the identity matrix; and $\mathbf{J}_f(\mathbf{X}^{(k)})$ is the interval Jacobian matrix at $\mathbf{X}^{(k)}$.

The following claims follow directly from the Mean Value Theorem:

- If $K(\mathbf{X}^{(k)}) \subset \mathbf{X}^{(k)}$, then $f(x) = 0$ has a unique solution for $x \in \mathbf{X}^{(k)}$.
- If $(K(\mathbf{X}^{(k)}) \cap \mathbf{X}^{(k)}) = \emptyset$, then $f(x) = 0$ has no solution in $x \in \mathbf{X}^{(k)}$.
- Otherwise, any solution of $f(x) = 0$ in $\mathbf{X}^{(k)}$ is in $K(\mathbf{X}^{(k)}) \cap \mathbf{X}^{(k)}$. Hence, for the next iteration $k + 1$, the Krawczyk method is repeated over the interval bounds $\mathbf{X}^{(k+1)} = K(\mathbf{X}^{(k)}) \cap \mathbf{X}^{(k)}$.

For more details, see [27, Thm. 8.2, Thm. 8.3], [26].

3.3 Linear Programming

Linear programming can be used as an optimization step in our approach. When functions can be bounded by convex regions over their arguments and results, linear programming can be used to derive tighter bounds or refute the existence of solutions. More generally, if a function $f(x)$ has no inflection points for the interval bounds for x , then computing a convex region that contains $f(x)$ vs. x is straightforward as shown in Figure 3.1. If we know the inflection points of $f(x)$,

Figure 3.1: Linear Constraints for $y = \tanh(-5 * x)$ for $-1 \leq x \leq 0$.

Blue curve indicates the original $\tanh$ function. The dashed lines are the linear constraints that are used. They are tangents at the boundaries of x and a secant line between the boundaries of x .

then we can compute convex regions for each interval between inflection points and then compute the convex hull of their union to obtain the desired linear program. When f is a function of one variable like in the case of $\tanh$, it is straightforward to calculate the convex hull of the union. For each region, we can compute the intersection points of our linear constraints (see Figure 3.1 for construction of linear constraints in a convex region). The convex hull of these intersection points from all regions form the convex hull of the union.

In our examples, however, there are functions expressed in terms of up to 3 variables. Computing the convex hull of the union is not as straightforward. Suppose there are two linear programs $LP1$ and $LP2$. In order to construct a union LP of $LP1$ and $LP2$, we can take each constraint $c1$ from $LP1$, and use its normal as

an objective function to solve $LP2$. This approach “moves” $c1$ if necessary so that $LP2$ is feasible with the moved $c1$. We repeat the process with constraints from $LP2$ as objective functions to solve $LP1$. The new set of “moved” constraints form LP - the union of $LP1$ and $LP2$. Figure 3.2 shows how a constraint is “moved” when taking the union. Taking the union of linear programs with this process involves solving multiple linear programs, which with our current method where an incremental linear programming solver is not used, is very slow. Hence, with functions with more than one variable, if interval bounds overlap inflection regions, I do not attempt to construct such a union. I only add the constraints indicating the interval bounds. Chapter 5 shows results where linear programming can provide significant performance and scalability advantages for some examples.

Figure 3.2: Moved Constraint in Union of Linear Programs.

Blue curve indicates the original $y = \tanh(-5 * x)$ function where $x = 0$ is the inflection point. The dashed lines indicate the linear constraints for the convex region defined by $-1 \leq x \leq 0$. This figure shows how the green secant constraint is moved when the union of linear programs for the convex regions on either side of the inflection point is taken.

Chapter 4

Implementation

This chapter describes the application of the Krawczyk method to find DC operating points of CMOS circuits. The core of our solver, which is written using Python, implements a bisection-based search of a hyper-rectangular region. Each hyper-rectangle is tested, first with interval-arithmetic evaluation of the system of equalities to be solved, and then with the Krawczyk method. Interval arithmetic evaluation can quickly refute infeasible hyper-rectangles, and the Krawczyk method can establish or refute the existence of a solution or further constrain the search hyper-rectangle. Section 4.1 describes this basic implementation; Section 4.2 adds linear-programming to the decision procedure; Section 4.3 describes implementation details about interval operations and interval evaluations of functions and gradients needed by the Krawczyk method; Section 4.4 describes the interval arithmetic models for CMOS circuits; and Section 4.5 describes possible extensions to our solver that might improve its efficiency but that I did not get a chance to explore in this work.

4.1 The Krawczyk Method and Bisection

As described in Chapter 3, the Krawczyk method is a variation of the interval Newton's method that can determine if a hyper-rectangle contains a unique solution, no solution or possibly more than one solution to a system of non-linear equalities. Bisection helps guide the search in the cases where the Krawczyk method cannot

resolve the status of a hyper-rectangle. This section describes various design trade-offs and decisions that I made when implementing the Krawczyk method. Because the Krawczyk method is an iterative process that is capable of reducing the size of the hyper-rectangle at each iteration, I needed to figure out the optimal number of times the Krawczyk step is performed before resorting to bisection. I also needed to take care of the case where the Krawczyk method fails to converge when the solution is close to the boundary of the hyper-rectangle. Furthermore, I explored two different ways of bisecting a hyper-rectangle. This section starts by describing the basic algorithm where bisection and the Krawczyk method are paired to find all hyper-rectangles each containing a unique DC operating point. Then it describes the decisions I made regarding the various options that I came across during the implementation.

For DC operating point analysis, we view the circuit as denoting a system of non-linear, ordinary differential equations. If x is the vector of node voltages, the circuit denotes a function f such that $\dot{x} = f(x)$. A DC operating point is a value for x such that $f(x) = 0$. I apply the Krawczyk method to find these values of x . My implementation first incorporates interval arithmetic evaluation of f . It checks if it can refute the region by testing $0 \notin F(\mathbf{X})$. A single, interval evaluation of f is faster than evaluating $\mathbf{J}_f(\mathbf{X})$ and the linear algebra functions for the Krawczyk method. Quickly eliminating the “easy” cases gave the implementation a significant speed improvement. The function implementing the Krawczyk method returns a status of ‘unique’, ‘unknown’, or ‘infeasible’. In the first two cases, it also returns a hyper-rectangle containing the solution or possible solutions. When a unique solution is found, it also adds the solution to the set of solutions.

Algorithm 1 depicts the basic approach incorporating the Krawczyk method with bisection. The solver takes an argument f , the function for which we are searching for all solutions for $f(x) = 0$ and $\mathbf{H}_0$, the hyper-rectangle in which to search. The basic reasoning behind the algorithm is to take care of the cases where the Krawczyk method cannot conclude if a hyper-rectangle $\mathbf{H}$ contains a unique solution or no solution. In this case, the solver can help direct the search by using bisection: choose a component of $\mathbf{H}$, and divide its interval into two segments to create two new hyper-rectangles, $\mathbf{H}_{lo}$ and $\mathbf{H}_{hi}$. The solver then tests each of these hyper-rectangles (recursively) for resolution with the Krawczyk method. This

Algorithm 1 Solver

```
1: procedure SOLVER( $f, \mathbf{H}_0$ )
2:   solutions  $\leftarrow \emptyset$ 
3:   workList  $\leftarrow$  new stack()
4:   workList.push( $\mathbf{H}_0$ )
5:   while not workList.empty() do
6:      $\mathbf{H} \leftarrow$  workList.pop()
7:      $[\text{status}, \mathbf{H}_K] \leftarrow$  Krawczyk( $f, \mathbf{H}$ ,
                                   solutions)
8:     if status == 'unknown' then
9:        $[\mathbf{H}_{lo}, \mathbf{H}_{hi}] = \text{split}(\mathbf{H}_K)$ 
10:      workList.push( $\mathbf{H}_{lo}$ )
11:      workList.push( $\mathbf{H}_{hi}$ )
12:    end if
13:  end while
14:  return solutions
15: end procedure
```

approach is similar to solvers such as GlobSol [19] that use interval verification methods like the interval Gauss-Seidel method. To the best of our knowledge the implementation presented in this work is the first to be used for DC operating point analysis, and likely the first to be used with an aim of integrating the method into formal hardware verification. Each time the hyper-rectangle is popped from the stack, the solver checks if it is contained in any of the existing hyper-rectangles in the solution set. If it is, then the solver does not need to consider it for the next steps any more.

The Krawczyk method is an iterative process and at each iteration it is possible for the Krawczyk step to reduce the size of hyper-rectangle under consideration. Thus, it is possible to have the solver repeatedly refine $\mathbf{H}$ until the status of the region is resolved or until it fails to decrease the volume of $\mathbf{H}$ by a specified factor. Experiments have shown that performing the Krawczyk step only once before resorting to bisection gives the smallest run-time. Table 4.1 shows example results of running the solver when I vary the number of times the Krawczyk step is performed before resorting to bisection. This experiment is done on a 4-stage Rambus ring oscillator (see Chapter 5, Section 5.2 for description of the Rambus

oscillator). The `modelType` indicates which model of inverters was used in the Rambus oscillator. `tanh` means that the inverter was modeled by a tanh function, `LCMOSFET` means that the inverter was modeled by 2 CMOS transistors with a long channel MOSFET model and `SCMOSFET` means that the inverter was modeled by 2 CMOS transistors with a short channel MOSFET model. I refer the reader to Section 4.4 at the end of this chapter for a more detailed description of the models. `gcc` indicates the ratio of strength of cross-coupled inverters to that of forward inverters in the Rambus ring oscillator. $k\alpha$ is the proportion of volume reduction threshold that defines how much the Krawczyk step should refine the volume of a hyper-rectangle before stopping. For example, if $k\alpha$ is 0.2, then the Krawczyk step will keep refining the hyper-rectangle in a single run as long as $\frac{\text{volumeAfterKrawczyk}}{\text{volumeBeforeKrawczyk}} < 0.8$. This means the smaller the value of $k\alpha$ the greater the number of times the Krawczyk step is performed. `#bisection` indicates the number of bisections. The total run-time is indicated in the `Total` column. With increasing $k\alpha$, even though the number of bisections increases, the total run-time generally decreases. The decreasing number of bisections with decreasing $k\alpha$ tells us that the Krawczyk method can significantly decrease the volume of spaces to be explored when it is applied more times before resorting to bisection. However, the expensive computation of Jacobian matrices and matrix products and inverses needed for the Krawczyk steps outweighs the advantage of further shrinking the search space. Thus, in my implementation we have $k\alpha$ set to 1.0. This means that the Krawczyk step is applied only once in one single call of the function. I use an epsilon inflation of 0.01 and a bisection procedure we call *bisectMax* for the experiments in Table 4.1. Epsilon inflation and bisection procedures are parameters that can be set in the solver and are explained in more details later in this section.

In some cases, the solver failed to converge when a solution was very close to the boundary of a hyper-rectangle. Even with repeated bisection, the Krawczyk update was not able to shrink $\mathbf{H}$ away from that boundary. I followed Caprini and Madsen’s method [6] method of tackling this problem as described by Rump [31, Sec. 10.5] by incorporating epsilon inflation in the Krawczyk method. The idea is to inflate the hyper-rectangle on either side in every dimension before applying the Krawczyk operator at each iteration. This accelerates convergence to a hyper-rectangle that the Krawczyk method can prove contains a unique solution. I use

Table 4.1: Variation of Run-time with Variations in Repetitions of the Krawczyk Step in Single Run for a 4-stage Rambus Oscillator. The `modelType` indicates which model of inverters was used in the Rambus oscillator. `tanh` means that the inverter was modeled by a tanh function, and `LCMOSFET` means that the inverter was modeled by 2 CMOS transistors with a long channel MOSFET model. `gcc` indicates the ratio of strength of cross-coupled inverters to that of forward inverters in the Rambus ring oscillator. $k\alpha$ is the proportion of volume reduction threshold that defines how much the Krawczyk step should refine the volume of a hyper-rectangle before stopping. `#bisect` indicates the number of bisections used in our method to solve the problem instance. The total run-time is indicated in the `Total` column.

<code>modelType</code>	<code>gcc</code>	$k\alpha$	<code>#bisect</code>	Total (s)
<code>tanh</code>	0.5	0.2	15401	70.178
<code>tanh</code>	0.5	0.4	15531	64.446
<code>tanh</code>	0.5	0.6	15749	62.105
<code>tanh</code>	0.5	0.8	16249	59.788
<code>tanh</code>	0.5	1.0	17619	55.171
<code>tanh</code>	4.0	0.2	10323	56.777
<code>tanh</code>	4.0	0.4	10479	52.822
<code>tanh</code>	4.0	0.6	10553	50.304
<code>tanh</code>	4.0	0.8	10623	50.938
<code>tanh</code>	4.0	1.0	13151	51.690
<code>LCMOSFET</code>	0.5	0.2	17494	543.256
<code>LCMOSFET</code>	0.5	0.4	18112	460.563
<code>LCMOSFET</code>	0.5	0.6	19204	410.711
<code>LCMOSFET</code>	0.5	0.8	21058	398.740
<code>LCMOSFET</code>	0.5	1.0	25602	378.229
<code>LCMOSFET</code>	4.0	0.2	3488	93.112
<code>LCMOSFET</code>	4.0	0.4	3603	85.161
<code>LCMOSFET</code>	4.0	0.6	3778	78.379
<code>LCMOSFET</code>	4.0	0.8	3897	77.215
<code>LCMOSFET</code>	4.0	1.0	5184	81.504

a smaller inflation size than that suggested by Rump - 0.1% of the distance of the hyper-rectangle on either side plus 10^{-20} rather than 10% plus 10^{-20} . The 10^{-20} term is added to make sure that there is always some kind of inflation no matter how tiny the size of the hyper-rectangle is. Table 4.2 shows results when the epsilon inflation is varied when the method is run on a 4-stage Rambus oscillator. With Rump's suggested inflation size (10%), the solver performs more bisections and has an increased run-time when compared to the smaller inflation sizes of 0.01%, 0.1% and 1%. The smaller inflation sizes require similar run-time in all the cases but an epsilon inflation of 0.01% requires a higher number of bisections than 0.1% for the tanh model when $gcc = 0.5$. I chose an epsilon inflation of 0.1%. It is possible we are observing these results because with larger hyper-rectangles resulting from the larger epsilon inflation, the interval Jacobian is big which makes it more difficult for the Krawczyk method to prove uniqueness. This leads to more bisections. A complication of using epsilon inflation is that the same solution can be found starting from adjacent regions. Before adding a solution to the return set, the solver checks if the new solution overlaps with any existing hyper-rectangles in the solution set. If so, the solver uses Newton's method to find a putative solution in the new hyper-rectangle. If the Newton's solution is within the intersection of the overlapping hyper-rectangles, the solver constructs a small hyper-rectangle around the Newton's solution and inside the intersection and use the Krawczyk method to confirm that the small hyper-rectangle contains a unique solution. This means that the new solution has already been found and does not need to be added. I use $k\alpha$ of 1.0 and the *bisectMax* bisection procedure for the experiments in Table 4.2.

The bisection procedure of the solver also needed some consideration. There are many ways to bisect a hyper-rectangle. I experimented with two different ways of bisection. In what we call the *bisectMax* method, the solver chooses the component with the largest interval to split over. In what we call the *bisectNewton* method, it uses a bisection method that is guided by Newton's method. In the *bisectNewton* method, the solver uses the Newton's method to find a solution in $\mathbf{H}$ with the mid point of $\mathbf{H}$ as the initial point. If Newton's method finds a solution in $\mathbf{H}$, the solver finds the component in $\mathbf{H}$ with the maximum distance between the Newton's solution and either border of the hyper-rectangle. This is the component that the solver chooses to bisect over. However instead of bisecting at the

Table 4.2: Variation of Run-time with Variation of Epsilon Inflation for a 4-stage Rambus Oscillator. The `modelType` indicates which model of inverters was used in the Rambus oscillator. `tanh` means that the inverter was modeled by a tanh function, and `LCMOSFET` means that the inverter was modeled by 2 CMOS transistors with a long channel MOSFET model. `gcc` indicates the ratio of strength of cross-coupled inverters to that of forward inverters in the Rambus ring oscillator. `epsilon (%)` indicates the proportion of hyper-rectangle interval length used for epsilon inflation. `#bisect` indicates the number of bisections used in our method to solve the problem instance. The total run-time is indicated in the `Total` column.

modelType	gcc	epsilon (%)	# bisect	Total (s)
tanh	0.5	0.01	17407	60.299
tanh	0.5	0.1	17189	54.653
tanh	0.5	1	17619	55.265
tanh	0.5	10	23443	73.882
tanh	4.0	0.01	12917	48.505
tanh	4.0	0.1	12925	46.760
tanh	4.0	1	13151	45.913
tanh	4.0	10	15329	60.240
LCMOSFET	0.5	0.01	23912	320.512
LCMOSFET	0.5	0.1	24064	321.110
LCMOSFET	0.5	1	25602	341.780
LCMOSFET	0.5	10	44873	633.321
LCMOSFET	4.0	0.01	5008	85.347
LCMOSFET	4.0	0.1	5022	77.057
LCMOSFET	4.0	1	5184	74.881
LCMOSFET	4.0	10	7219	95.318

midpoint, it bisects so that one hyper-rectangle $\mathbf{H}_{lo}$ entirely contains the Newton's solution with the solution significantly away from the borders and the other $\mathbf{H}_{hi}$ does not. Figure 4.1 shows how the two bisection procedures work. The idea is to help the Krawczyk method quickly prune search spaces and find hyper-rectangles containing the unique solution. If the Newton's method cannot find a solution in $\mathbf{H}$, then the solver just uses the *bisectMax* method to bisect. My experiments have shown that the *bisectNewton* method requires fewer bisections than *bisectMax* for

Figure 4.1: Bisection Procedures

the tanh model as shown in the results for a 4-stage Rambus ring oscillator in Table 4.3. For *bisectNewton* method to be useful in terms of run-time, the advantage gained by the decrease in number of bisections must outweigh the disadvantage of performing the extra Newton's step each time we bisect. For the tanh and long channel MOSFET case, we do not get run-time improvement. As far as the short channel MOSFET model is concerned, there is a decrease in the number of bisections when *bisectNewton* is used rather than *bisectMax*. However the proportion of decrease is small compared to that of the tanh case and the long channel case when $gcc = 4.0$. I do not have a reason for this result yet. The run-time for *bisectNewton* is further increased because computing gradients for the more complicated short channel MOSFET model is very expensive when compared to the other models. For the experiments in Chapter 5, I used *bisectMax* for all the models. I used a $k\alpha$ of 1.0 and epsilon inflation of 0.1% for the experiments in Table 4.3.

4.2 Linear Programming as a Decision Procedure

The basic combination of the Krawczyk method and bisection leads to the exploration of the search space in a systematic way. The interval formulation in our solver makes it suitable to incorporate linear programming as a decision procedure. The user gets the option to choose linear programming because experiments have shown that linear programming helps only with certain models in our current

Table 4.3: Comparison of Bisection Types for a 4-stage Rambus oscillator. *tanh* means that the inverter was modeled by a *tanh* function, *LCMOSFET* means that the inverter was modeled by 2 CMOS transistors with a long channel MOSFET model, and *SCMOSFET* means that the inverter was modeled by 2 CMOS transistors with a short channel MOSFET model. *gcc* indicates the ratio of strength of cross-coupled inverters to that of forward inverters in the Rambus ring oscillator. *bisectType* indicates the type of bisection used in our method. *#bisect* indicates the number of bisections used in our method to solve the problem instance. The total run-time is indicated in the *Total* column.

modelType	<i>gcc</i>	bisectType	# bisect	Total (s)
tanh	0.5	bisectMax	17189	59.357
tanh	0.5	bisectNewton	12297	68.821
tanh	4.0	bisectMax	12925	48.354
tanh	4.0	bisectNewton	9239	41.698
LCMOSFET	0.5	bisectMax	24064	340.764
LCMOSFET	0.5	bisectNewton	19548	458.364
LCMOSFET	4.0	bisectMax	5022	77.749
LCMOSFET	4.0	bisectNewton	4331	79.493
SCMOSFET	4.0	bisectMax	17910	337.586
SCMOSFET	4.0	bisectNewton	16347	593.621

framework (see Chapter 5 for more details). This section describes how linear program fits into our solver should the user choose to turn on the linear programming feature.

Linear programming can refute an infeasible hyper-rectangle and if it is not infeasible, then tighten the bounds on a hyper-rectangle. Applying linear programming on a hyper-rectangle means adding constraints relevant to the problem defined by function f constrained within the hyper-rectangle to a linear-programming solver and then solving it. Chapter 3 Section 3.3 describes how we can construct such constraints for a specific hyper-rectangle. If the linear programming step deems the hyper-rectangle infeasible then the solver is finished with that hyper-rectangle. Otherwise the solver tightens the bounds on the hyper-rectangle by adding objective functions minimizing and maximizing each variable in f and then solving the respective linear programs. I used CVXOPT [1] to solve linear

Algorithm 2 Algorithm with LP

```
1: procedure SOLVER( $f, \mathbf{H}_0$ )
2:   solutions  $\leftarrow \emptyset$ 
3:   [status1,  $\mathbf{H}_1$ ]  $\leftarrow$  Krawczyk( $f, \mathbf{H}_0$ , solutions)
4:   if status1  $\neq$  'unknown' then
5:     return solutions
6:   end if
7:   workList  $\leftarrow$  new stack()
8:   workList.push( $\mathbf{H}_1$ )
9:   while not workList.empty() do
10:     $\mathbf{H} \leftarrow$  workList.pop()
11:    [status,  $\mathbf{H}_{LP}$ ]  $\leftarrow$  lp( $f, \mathbf{H}$ )
12:    if lpStatus == 'feasible' then
13:       $\mathbf{H} \leftarrow \mathbf{H}_{LP}$ 
14:    else
15:      continue
16:    end if
17:    [status,  $\mathbf{H}_K$ ]  $\leftarrow$  Krawczyk( $f, \mathbf{H}$ ,
                                solutions)
18:    if status == 'unknown' then
19:      bisect( $f, \mathbf{H}_K$ , worklist, solutions)
20:    end if
21:  end while
22:  return solutions
23: end procedure
```

programs in the solver because CVXOPT is an open source convex optimization package that fit easily in my Python framework. While CVXOPT is suitable for a proof-of-concept implementation, we believe that replacing it with an incremental linear programming solver would greatly improve the run-time. With CVXOPT, we need to restart the linear program from scratch every time we are solving for the same set of constraints with a different objective function. We hypothesized that an incremental linear programming solver will improve run-time efficiency by avoiding this restarting process. This is an area of improvement that I did not get to explore in this research work.

Algorithm 2 describes how linear programming fits into our solver. It takes in the same parameters as the basic bisection and the Krawczyk method (see Algo-

rithm 1). This procedure maintains a stack containing hyper-rectangles that have been tested using the Krawczyk method and have status ‘unknown’. When a hyper-rectangle is popped from the stack, the solver first uses linear programming to check its feasibility. If it is infeasible, then the solver is done with that hyper-rectangle. Otherwise, it tightens the bounds using linear programming by setting objective functions to minimize and maximize each variable. The smaller hyper-rectangle, $\mathbf{H}_{LP}$ is then tested with the Krawczyk operator. If the status of the hyper-rectangle is still unresolved, then the solver proceeds with our specialized bisection procedure.

Algorithm 3 Bisection procedure

```

1: procedure BISECT( $f, \mathbf{H}$ , worklist, solutions)
2:   while True do
3:      $[\mathbf{H}_1, \mathbf{H}_2] = \text{split}(\mathbf{H})$ 
4:      $[\text{status1}, \mathbf{H}_{K,1}] \leftarrow \text{Krawczyk}(f, \mathbf{H}_1, \text{solutions})$ 
5:      $[\text{status2}, \mathbf{H}_{K,2}] \leftarrow \text{Krawczyk}(f, \mathbf{H}_2, \text{solutions})$ 
6:     if status1 == ‘unknown’ then
7:       if status2 == ‘unknown’ then
8:         worklist.push( $\mathbf{H}_{K,1}$ )
9:         worklist.push( $\mathbf{H}_{K,2}$ )
10:      return
11:     else
12:        $\mathbf{H} = \mathbf{H}_{K,1}$ 
13:     end if
14:     else if (status2 == ‘unknown’) then
15:        $\mathbf{H} = \mathbf{H}_{K,2}$ 
16:     else
17:       return
18:     end if
19:   end while
20: end procedure

```

Algorithm 3 shows the bisection procedure. We describe the approach as “bisect-and-kill”. After splitting a hyper-rectangle, the bisect procedure uses the Krawczyk method to determine if the status of either or both of the smaller hyper-rectangles can be resolved. If one can be resolved, the bisect procedure continues by splitting the other. If both are resolved, it is done with that region, and bisect

returns. If neither can be resolved, they are both pushed onto the stack for further analysis. This approach implements a trade-off between bisection, which is fast but can produce an exponential explosion in the number of hyper-rectangles to analyze, and the linear-programming tests which are slower but do not increase the number of regions to explore. As long as bisection is not increasing the number of regions under consideration, the procedure continues bisecting. When a bisection results in two, unresolved regions, it performs the more expensive analysis.

4.3 Interval Arithmetic Implementation Details

Chapter 3, Section 3.2 showed how the Krawczyk operator requires interval operations and interval evaluation of functions and Jacobians. This section describes the rounding method I used in the implementation of the Krawczyk method to maintain rigorous interval bounds and the Python packages I used for interval evaluation of functions and gradients.

It is necessary to do outward interval rounding where we increase the upper bound of the interval and decrease the lower bound of the interval by some amount after each interval operation in the implementation of the Krawczyk operator to maintain rigorous interval bounds. The interval upper bound rounding is done towards $+\infty$ mode where one unit of least precision (*ulp*) is added to the upper bound. The interval lower bound rounding is done towards $-\infty$ mode where one *ulp* is subtracted from the lower bound. In [31, Chapter 4], Rump describes directed rounding for rigorous interval bounds. In directed rounding, the processor is first set to the round towards $-\infty$ mode after which all the subsequent operations involving lower bounds of intervals are done in a round down mode. The processor is then set to the round towards $+\infty$ mode after which all the subsequent operations involving upper bounds of intervals are done in round up mode. While directed rounding gives a faster and more efficient way of rounding in interval operations, its implementation would require special code to set the floating point hardware rounding modes and to separate out all the interval operations into a round down group and round up group and then merge the evaluations together. I chose a simpler way of interval rounding as suggested by Moore in [26]. I add one *ulp* to the upper bound and subtract one *ulp* from the lower bound of an interval at every in-

terval operation rather than separating out the round down and round up operations as in directed rounding.

A naive implementation of such rounding would make computations like interval matrix multiplication required in the Krawczyk operator very expensive. For example, matrix multiplication would involve writing a triple `for`-loop. To improve performance, it is important to take advantage of `numpy`'s matrix/vector manipulation routines as much as possible. `Numpy` uses BLAS routines under the hood that makes matrix operations much faster. In order to achieve this, I represented an interval $\mathbf{X}$ as its mid point and its radius. Specifically,

$$\mathbf{X} = [x_{lo}, x_{hi}] = [x_m - x_r, x_m + x_r] = (x_m, x_r) \quad (4.1)$$

where x_m is the midpoint of $\mathbf{X}$ and x_r is the radius and is always non-negative. Multiplication of two intervals $\mathbf{X}, \mathbf{Y}$ gives us:

$$\mathbf{X} * \mathbf{Y} = (x_m, x_r) * (y_m, y_r) = (x_m * y_m, |x_m| * y_r + |y_m| * x_r + x_r * y_r) \quad (4.2)$$

Just like an interval, we can also represent an interval matrix as a mid point matrix and a radius matrix. Multiplication of interval matrices A , and B involves multiplying the mid point matrices A_m, B_m , to get the mid point matrix C_m of the result matrix C . The radius of C , C_r equals $|A_m| * B_r + |B_m| * A_r + A_r * B_r$. Conservative interval rounding for all the operations involved in the matrix multiplication is done by doing element wise addition of $0.5 * (n + 1) * U$ to C_r , where $U = (\text{ulp}(|A_m| * |B_m|) + \text{ulp}(C_r))$ and n is the number of operations performed in calculating each element of C . This is because doing outward rounding at each step of the matrix multiplication creates a total outward rounding of at most $n * U$. However, because floating point hardware does “round nearest”, the actual rounding is at most $0.5 * n * U$ but we bound it to $0.5 * (n + 1) * U$ in case $0.5 * U$ is too small to be represented by the machine floating point number. Interval matrix and vector multiplications can also be done in a similar way. Interval matrix addition or subtraction can be done using `numpy`'s existing method where intervals are represented in the form of an array of lower and upper bound. Interval rounding is done by subtracting and adding `ulp` to the resulting matrix. Taking advantage of

numpy's existing matrix operations brought about a significant improvement in the solver performance.

The Krawczyk method also requires interval evaluation of functions and their Jacobians. The INTLAB package for MATLAB combines interval arithmetic and automatic differentiation. However, there are no such packages in Python. My solution was to use a C++ automatic differentiation package called FADBAD++ [4]. FADBAD++ can accept classes other than floating points as long as the classes define specific operations including addition, subtraction, multiplication. Rather than writing my own interval class, I decided to build upon an already existing interval class from the C++ MC++ package [2]. I modified the MC++ interval class by incorporating interval rounding with *ulp*. These packages made it easier to calculate Jacobian interval for functions for which manually calculating gradients is difficult (see the short channel MOSFET model in Section 4.4). I created Python bindings for the C++ code using the boost library [32] so that they can be incorporated in our solver. Specifically, I used the boost Python library to export the C++ classes implementing the interval functions and gradients to Python.

4.4 Circuit Models

The circuit modelling approach used for the solver is based on “nodal analysis” [16]. A circuit is a collection of components; each component has some (fixed) set of terminals; connected terminals form a node. The *state* of a circuit is an assignment of voltages V , to the nodes. Each component has a function calculating the electrical current I_M that flows out of the component through each terminal based on the voltages on the nodes of those terminals. Furthermore, components can define capacitances between their terminals, and these capacitances can be functions of the node voltages. If we let C be the matrix denoting the internode capacitances and I_C be the current due to capacitances, then the time derivative of V , $\dot{V}$ is defined by

$$I_C = -C\dot{V} \quad (4.3)$$

According to Kirchoff's current law:

$$I_C + I_M = 0 \quad (4.4)$$

At a DC operating point, $\dot{V} = 0$, and because C is structurally non-singular, $I_C = 0$ from Equation 4.3 which leads to $I_M = 0$ from Equation 4.4. Thus to find DC operating points, we need to find the voltages for which the total current flowing into each node from the components is zero.

This section gives a description of functions calculating current from components in terms of node voltages using three different models — tanh function to model an inverter, a long channel MOSFET model [36], and a short channel MOSFET model [23] for a transistor. I introduced the tanh model in the initial stages of this work to provide a simple way to test our method. The long channel MOSFET model can be used for comparison purposes with other solvers such as Z3 as it can be described using piece-wise polynomials. I used the short channel MOSFET model because it is a more realistic model for transistors. Additionally, the parameters for the short channel MOSFET model were fit by a colleague in our lab so I had in-house support and working code to get the short channel MOSFET model working. The tanh model and the long channel MOSFET models were simple enough that I could perform all the gradient calculations manually. The short channel MOSFET model, on the other hand, required automatic differentiation which inspired us to incorporate FADBAD++ and MC++ in our solver as described in Section 4.3. The rest of this section gives more details about the implementation of interval evaluations of the models and the subsequent gradients.

4.4.1 tanh Inverter Model

In this model, an inverter has a transfer function given by a tanh function. In particular:

$$I_{out} = g * \tanh(a * V_{in}) - V_{out} \quad (4.5)$$

where $a < 0$ is the inverter gain (we use $a = -5$), and g models the output conductance. Because tanh is monotonic, it is enough to take the tanh at the end points for an interval evaluation of the function. Similarly the gradient of tanh is also monotonic on either side of the inflection point which is at $V_{in} = 0$. If interval V_{in} does

not contain the inflection point, evaluating the gradients at the end points gives the interval gradient. If it does then we need to take the gradient at the inflection point in addition to the end points and take the minimum and maximum of these three gradients to get the interval gradient.

4.4.2 Long Channel MOSFET Transistor Model

I also modelled a CMOS transistor with a long channel MOSFET model from Zaki *et al.* [36]. Given drain voltage Vd , gate voltage Vg , and source voltage Vs , the model expresses Ids , current from drain to source in terms of Vds and Vgs where $Vds = Vd - Vs$ and $Vgs = Vg - Vs$. The following shows how Ids is calculated for an nfet when $Vd \geq Vs$.

$$Ids = \begin{cases} 0 & Vgs \leq Vt(cutoff) \\ 0.5 * SK * (Vgs - Vt)^2 & Vds \geq Vgs - Vt(saturation) \\ SK * (Vgs - Vt - 0.5 * Vds) * Vds & Vds \leq Vgs - Vt(linear) \end{cases} \quad (4.6)$$

where K is $270 * 10^{-6} \text{Amp/Volt}^2$ for nfet and $-90 * 10^{-6} \text{Amp/Volt}^2$ for pfet; Vt is 0.4 for nfet and -0.4 for pfet. nfet has a shape factor S of 3.0 and pfet has a shape factor S of 6.0. The power is 1.8V. Like the tanh model, this long channel MOSFET model is also monotonic, so I calculated the interval extension of Ids , **Ids** by taking the function values at the end points of **Vd**, **Vg**, **Vs**. For example, for an nfet, **Ids_{lo}** occurs at (**Vd_{lo}**, **Vg_{lo}**, **Vs_{hi}**) and **Ids_{hi}** occurs at (**Vd_{hi}**, **Vg_{hi}**, **Vs_{lo}**) when **Vd** $\geq$ **Vs**. When **Vd** and **Vs** do not overlap, it is straightforward to manually calculate the interval gradient from the Ids function given above. However, when **Vd** and **Vs** do overlap, we need to calculate the gradients separately for the case when **Vds** < 0 and **Vds** ≥ 0 and take the union of the resulting intervals. Repeated calls to the function calculating the gradient slow down the performance. When bisecting on a voltage interval, many transistors will have no changes to the voltage intervals for their terminals. By memoizing the results of transistor model evaluation, the solver obtained a significant improvement in performance over a direct implementation for the long channel model.

4.4.3 Short Channel MOSFET Transistor Model

The short channel MOSFET model [23] calculating current from drain to source I_{ds} cannot be expressed with a few lines of equations. It has about 20 parameters, and I refer the reader to Appendix A.1 for the C++ source code describing the model. As mentioned before, I used the C++ automatic differentiation package FADBAD++ and interval class from MC++ package to implement this short channel model. Repeated calls to the C++ function calculating the gradient using the automatic differentiation package slow down the performance. Just like in the case of the long channel MOSFET model, memoizing the gradients gave significant improvement in performance for the short channel MOSFET model.

To make automatic differentiation from FADBAD++ work appropriately, I needed to define a function *mvs_Id* that takes in interval drain, source and gate voltages indicated by V_d , V_s and V_g respectively as parameters and return interval current from drain to source I_{ds} as calculated according to the short channel model (see Appendix A.1). In this function, each arithmetic operation is replaced with its interval arithmetic equivalent using the MC++ package. For an nfet, the body voltage V_b is set to 0.0 and for a pfet it is set to the power which is 1.0. The short channel MOSFET model is monotonic just like the long channel MOSFET model. Therefore, in order to calculate I_{ds} , it is enough to take the values at two corners of a hyper-rectangle. This is different from the I_{ds} computation in *mvs_Id* where each arithmetic operation in the evaluation is replaced with its interval arithmetic equivalent. Evaluation at the corners gave tighter current bounds and significant performance improvement in terms of run-time. Chapter 5 Section 5.2 shows results where I can evaluate the 4 stage Rambus Ring oscillator with the short channel MOSFET model using our solver. Without taking advantage of the monotonicity properties, our solver timed out on the same example. For the gradients when V_d and V_s do not overlap, I let FADBAD++ calculate the interval gradients from our interval defined function *mvs_Id*. If they do overlap, just like in the long channel MOSFET case, I took care of $V_{ds} < 0$ and $V_{ds} \geq 0$ cases separately using FADBAD++ and then took the union of the resulting intervals as the overall gradient.

4.5 Possible Extensions

As mentioned in the previous sections there are various way that the implementation of our method can be improved - for example by using directed rounding or using an incremental linear programming solver. Additionally it is possible that modifications to our decision procedure can help significantly improve the performance. This section describes such modifications to be explored further in future work.

Table 4.1 shows that even though repeated iterations of the Krawczyk step decreases the number of hyper-rectangles to be explored, the run-time does not improve. This is because computations of the interval Jacobians and linear algebra functions involved in the Krawczyk method computation is expensive. One possible modification of our solver that could tackle this problem is to limit the usage of the Krawczyk method to only when it is needed. We could incorporate the same “bisect and kill” approach that I took in Section 4.2. Except this time, we would deal with a combination of interval arithmetic and the Krawczyk method. After splitting a hyper-rectangle, the bisect procedure would use the interval evaluation of the function to test $0 \notin f(\mathbf{X})$ to check if either or both of the smaller hyper-rectangles can be refuted. If one can be refuted, the bisect procedure would continue by splitting the other. If both are refuted, it is done with that region, and bisect would return. If neither can be refuted, they would both be pushed onto the stack for further analysis by the Krawczyk method. This approach would implement a trade-off between interval arithmetic evaluation of the function, which is fast but cannot prove existence of unique solutions, and the Krawczyk method which is slower but can prove uniqueness in addition to refuting and shrinking a hyper-rectangle. As long as bisection is not increasing the number of regions under consideration, the procedure would continue bisecting. When a bisection results in two, unrefuted regions, it would perform the more expensive analysis.

Chapter 5

Results

I tried our solver on two main sets of examples: finding DC operating points for the Rambus ring oscillator problem [17] and for a Schmitt trigger circuit. I compared the results of the solver described in this paper with dReal [11] and the Z3 [8] SMT solvers.

As described in Chapter 4, Section 4.4, I used 3 different models for our simplest circuit components — tanh to model inverters, a long channel MOSFET model, and a short channel MOSFET model for CMOS transistors. dReal is able to handle all these models. I do not model the short channel MOSFET model in Z3 because Z3 cannot handle transcendental functions. Due to the simple nature of the tanh model, we can use it in Z3 by approximating it as a piecewise polynomial function:

$$\tanh(ax) \approx \begin{cases} -1 - (ax)^{-5} & ax \leq -2 \\ -\frac{13}{256}(ax)^3 + \frac{11}{16}(ax) & -2 < ax < 2 \\ 1 - (ax)^{-5} & ax \geq 2 \end{cases} \quad (5.1)$$

Figure 5.1 shows the comparison between the piece-wise approximation with the actual tanh function.

Our approach finds all the DC operating points. For comparison purposes, I also used dReal and Z3 to find all the DC operating points. I achieved this by finding one DC operating point at a time and then adding constraints to exclude

Figure 5.1: Approximation of $\tanh(a * x)$ where $a = -5$ with a piece-wise polynomial function

the solution. Because in Z3, the solution is in the form of an algebraic number, I added the constraint to exclude the exact solution. In dReal however, I excluded a small region around the solution. This is unsound because the exclusion region could accidentally enclose another solution, but dReal does not provide a way to enumerate all solutions. If too small an exclusion region is used then dReal might get stuck finding the same solution. I used $\delta = 10^{-14}$ for our dReal experiments where δ represents the perturbation of the original problem. For the tanh and the short channel MOSFET model case, an exclusion region of $4 * 10^{-14}$ was enough to find all relevant solution regions in the experiments that did not time-out. However for the long channel MOSFET case, an exclusion region of $2 * 10^{-11}$ was needed.

I used the best possible configurations for our solver when comparing it with dReal and Z3. As described in Chapter 4, this means that the solver uses an epsilon inflation of 0.1%; it performs the Krawczyk step only once in a single call; it uses

Table 5.1: Inverter – comparison of our algorithm and dReal and Z3

Model	# Input Voltage	Time – our method (s)	Time – dReal (s)	Time – Z3 (s)
tanh	-1.0	0.006	0.002	0.037
tanh	1.0	0.005	0.002	0.033
LCMOSFET	0.0	0.012	0.005	0.060
LCMOSFET	1.8	0.016	0.005	0.044
SCMOSFET	0.0	0.029	0.446	-
SCMOSFET	1.0	0.026	0.441	-

bisectMax for all the models. All times are reported for executions on a MacBook Air with a 1.6GHz Intel i5 processor and 4 GB memory. The run-time results in the rest of this chapter are shown in tables. An entry of “time-out” in the run-time column means that the solver did not complete in 10 hours. An entry of “incomplete” means that the solver found some, but not all solutions before reaching the 10 hour time out and an entry of “prior time-out” means that the solver timed out on a smaller version of the problem; so we did not try the larger one. To show scalability advantages of our tool as compared to dReal and Z3, this chapter first shows results for very simple circuits involving inverters and then moves on to the main examples involving the Rambus ring oscillator and the Schmitt trigger.

5.1 Simple Circuits

In the simplest experiment, I took an inverter and found the output voltage for a specific input voltage at the DC operating point. I compared our method of finding the output voltage with dReal and Z3. For these experiments, I did not use linear programming in our method. Table 5.1 shows the results. In all the cases, our solver performed better than Z3. For the tanh case, our solver had comparable runtimes with dReal. For the long channel MOSFET case, dReal performed better than our solver. However, for the short channel MOSFET case, our solver outperformed dReal.

Slightly more complicated circuits involving inverter loops are shown in Figure 5.2. Table 5.2 shows the results of our experiments where our solver was used to find the DC operating points of these circuits. Our solver performed better than Z3 in all cases. In the case of the tanh model, dReal gave a better performance

(a) Inverter Loop with 1 Inverter

(b) Inverter Loop with 2 Inverters

(c) Inverter Loop with 3 Inverters

(d) Inverter Loop with 4 Inverters

Figure 5.2: Inverter Loops

than our solver. For the long channel MOSFET model in the simpler circuits (with 1 and 2 inverters), dReal performed better than our solver. When the number of inverters was increased, our solver was significantly faster than dReal. Our solver performed better than dReal for all the short channel cases.

Hence even though our solver is slower than dReal in simpler cases, it shows better scaling for these examples.

5.2 Rambus Ring Oscillator

Jones *et al.* [17] presented the Rambus Ring oscillator as a challenge problem. It consists of an even number of stages connected in a Möbius topology. Figure 5.3

Table 5.2: Inverter Loop – comparison of our algorithm and dReal and Z3

Model	# inverters	# solutions	Time – our method (s)	Time – dReal (s)	Time – Z3 (s)
tanh	1	1	0.003	0.004	0.054
tanh	2	3	0.013	0.011	0.615
tanh	3	1	0.014	0.005	10.456
tanh	4	3	0.036	0.013	time-out
LCMOSFET	1	1	0.014	0.007	0.292
LCMOSFET	2	3	0.107	0.098	0.300
LCMOSFET	3	1	0.074	0.484	0.968
LCMOSFET	4	3	0.271	11.579	25.829
SCMOSFET	1	1	0.037	4213.660	-
SCMOSFET	2	3	0.248	incomplete	-
SCMOSFET	3	1	0.415	time-out	-
SCMOSFET	4	3	0.967	prior time-out	-

Figure 5.3: Rambus Ring Oscillator

shows a 2 stage ring oscillator. The Rambus researchers reported that examples of this design had failed intermittently in their test lab. The problem was known to be related to the inverter strengths (i.e. transistor sizes) in the circuit design. When the forward inverters (labeled as fwd) are much stronger than the cross-coupled inverters (labeled as cc), the n stage oscillator essentially functions as a ring for $2n$ inverters. The circuit has two stable DC operating points corresponding to consistent truth assignments to the outputs of the forward inverters while ignoring the cross-coupled ones. It also has one unstable operating point when all inverter outputs are at the voltage where the inverter input voltage equals the output voltage for the DC transfer function of the inverter. If the cross-coupled inverters are much larger than the forward ones, then the circuit has 3^n operating points, of which 2^n are stable. Designers typically want to make the cross-coupled inverters as small as possible – this allows the circuit to operate at a higher frequency and use less power. If the cross-coupled inverters are too small, the oscillator can fail to start due to the stable operating points described above. If the cross-coupled inverters are only slightly too small, then failures will only occur for rare initial conditions. This rare but potential fatal failure motivates using formal approaches to verification.

Table 5.3 shows the results for the Rambus oscillator example. In this version, I did not use linear programming. The entries for gcc are the ratio of the size of the cross-coupled inverters divided by the size of the forward inverters. If $gcc = 0.5$, an n -stage oscillator should have 3 operating points. If $gcc = 4.0$, then there should be 3^n operating points. We didn't know the 3^n result before using our tool for the analysis. Once we saw the pattern, we found a simple proof (see Appendix A.2). This confirms that our tool has found all solutions, and it shows the value of this analysis for better understanding the behaviour of the circuit.

dReal could solve the smaller circuits with the tanh model significantly faster than our method. In the case of larger number of stages and more operating points, our solver outperformed dReal. For the MOSFET models, dReal could only solve one example, and it was much slower than our solver. For the long channel MOSFET model with two stages and $gcc = 4.0$, dReal found all 9 operating points within 10 hours, but was unable to verify that there were no further solutions before the time-out. For the short channel MOSFET model, dReal timed out in all

Table 5.3: Rambus Ring Oscillator – comparison of our algorithm and dReal

Model	# Stages	<i>gcc</i>	# solutions	Time – our method (s)	Time – dReal (s)
tanh	2	0.5	3	0.921	0.075
tanh	2	4.0	9	0.593	0.342
tanh	4	0.5	3	52.763	0.480
tanh	4	4.0	81	44.663	incomplete
tanh	6	0.5	3	2458.652	time-out
tanh	6	4.0	729	3496.199	prior time-out
LCMOSFET	2	0.5	3	5.722	4935.832
LCMOSFET	2	4.0	9	1.948	incomplete
LCMOSFET	4	0.5	3	324.251	time-out
LCMOSFET	4	4.0	81	73.316	prior time-out
LCMOSFET	6	0.5	3	18400.544	prior time-out
LCMOSFET	6	4.0	729	2951.956	prior time-out
SCMOSFET	2	0.5	3	16.166	time-out
SCMOSFET	2	4.0	9	5.439	time-out
SCMOSFET	4	0.5	3	556.849	prior time-out
SCMOSFET	4	4.0	81	285.650	prior time-out
SCMOSFET	6	0.5	3	24353.846	prior time-out
SCMOSFET	6	4.0	729	12567.220	prior time-out

the cases. Our method clearly scales to larger problem instances for this oscillator analysis problem.

Z3 found the solution where all voltages are zero for all instances of the oscillator problem with the tanh model, and then timed out before finding any of the other solutions.

Table 5.4 gives a more detailed breakdown of the run-times. The columns #-kill1, #-kill2 and LP time are only relevant in the cases when linear programming was used. The column #-kill1 reports the number of times that one hyper-rectangle produced by a bisection was immediately classified by the Krawczyk method as having a unique solution or having no solutions. The column #-kill2 reports the number of times that both hyper-rectangle produced by a bisection were immediately classified. The column LP-time reports the total time spent using linear programming to check the feasibility of hyper-rectangles and, if feasible, tighten their bounds. The columns #-K and K-time indicate the number of calls and the total time spent evaluating the Krawczyk operator.

When the linear-programming methods were used to tighten bounds, the number of regions explored dropped dramatically for the tanh model which gave an im-

Table 5.4: Rambus Ring Oscillator – runtime detail. Linear programming for short channel MOSFET model was not implemented and therefore the short channel MOSFET model results are not included.

Model	# Stages	<i>gcc</i>	# bisect	#-kill1	#-kill2	# K	K-time (s)	LP-time (s)	Total time (s)
tanh w. LP	2	0.5	27	6	0	86	0.128	0.608	0.743
tanh w. LP	2	4.0	29	20	0	78	0.096	0.396	0.504
tanh w. LP	4	0.5	67	6	4	214	0.553	4.305	4.875
tanh w. LP	4	4.0	278	115	15	812	1.918	13.112	15.209
tanh w. LP	6	0.5	209	43	15	623	2.259	21.151	23.452
tanh w. LP	6	4.0	3402	1775	76	9317	32.047	316.679	356.661
tanh no LP	2	0.5	473	—	—	947	0.872	—	0.921
tanh no LP	2	4.0	305	—	—	611	0.546	—	0.593
tanh no LP	4	0.5	17189	—	—	34379	51.296	—	52.763
tanh no LP	4	4.0	12925	—	—	25851	39.656	—	44.663
tanh no LP	6	0.5	570503	—	—	1141007	2417.303	—	2458.652
tanh no LP	6	4.0	486119	—	—	972239	2129.592	—	3496.199
LCMOSFET w. LP	2	0.5	444	157	119	1222	6.051	7.727	13.862
LCMOSFET w. LP	2	4.0	130	68	22	339	1.853	1.791	3.681
LCMOSFET w. LP	4	0.5	14355	7434	5047	53441	397.885	1089.534	1491.445
LCMOSFET w. LP	4	4.0	2550	1967	939	10809	79.511	137.153	218.437
LCMOSFET no LP	2	0.5	545	—	—	1091	5.650	—	5.722
LCMOSFET no LP	2	4.0	163	—	—	327	1.914	—	1.948
LCMOSFET no LP	4	0.5	24064	—	—	48129	321.851	—	324.251
LCMOSFET no LP	4	4.0	5022	—	—	10045	70.840	—	73.316

Figure 5.4: Schmitt Trigger.

proved run-time performance. However, the linear-programming operations were much slower than evaluations of the Krawczyk operator. In part, this is because for a problem with m variables, we solve $2m$ linear programming problems to derive tighter bounds for each linear-programming step. Each of these start the linear program solver from scratch. This is a consequence of using the linear-programming solver from CVXOPT package. For the long channel MOSFET model, the decrease in the number of regions explored was very small compared to the decrease seen with the tanh model. One of the possible reasons is that I used linear constraints to bound each transistor instead of an inverter like the tanh model. Tighter linear constraints and an incremental linear programming solver are promising areas of further research. Since we did not observe promising results with the linear programming for the long channel MOSFET model with our current framework, I did not implement it for the short channel MOSFET model.

Figure 5.5: DC Operating Points of Schmitt Trigger using the short channel MOSFET model

5.3 Schmitt Trigger

Figure 5.4 shows an inverting Schmitt trigger from [13, p. 283]. The circuit exhibits hysteresis. For inputs (V_{in} in Figure 5.4) that are sufficiently low, the output ($V(0)$ in Figure 5.4) will be high, and for inputs that are sufficiently high, the output will be low. Between these two, there is a range of input voltages for which the output can settle high or low, depending on its history. For such voltages, there are actually three operating values for the output: the stable low and high values, and an unstable operating value in between these two. Figure 5.5 shows output voltages at DC operating points for specific input voltages in a Schmitt trigger using the short channel MOSFET model.

Like the Rambus oscillator example, I used the long channel MOSFET model and parameters from [36] and the short channel MOSFET model from [29]. A

Table 5.5: Schmitt Trigger – comparison of our algorithm and dReal

Model	# Input Voltage	#Solutions	Time – our method (s)	Time – dReal (s)
LCMOSFET	0.0	1	0.273	0.185
LCMOSFET	0.9	3	0.694	4.141
LCMOSFET	1.8	1	0.210	0.167
SCMOSFET	0.0	1	0.509	time-out
SCMOSFET	0.5	3	1.869	prior time-out
SCMOSFET	1.0	1	0.468	prior time-out

weakness of our solver is that the Krawczyk method fails to converge if the Jacobian at the solution is a singular matrix. Singular Jacobians structurally arise in the case of the Schmitt trigger with the long channel MOSFET model. In order to avoid this issue, I added a leakage term in the form of $10^{-8} * (Vd - Vs)$ to the current function where Vd and Vs are drain and source voltages respectively.

I used our solver, dReal, and Z3 to search for these DC operating points. As with the Rambus oscillator examples, Z3 timed out for all the long channel MOSFET model cases. Run-times for our solver and dReal are shown in Table 5.5. In these experiments, I did not use linear programming in our method.

To illustrate a further application of our DC operating point analysis, I analyzed the effectiveness of the Schmitt trigger for resolving metastability. The operating point analysis solves for $I = f(V) = 0$. The Jacobian of f characterizes the operating point: if all eigenvalues have negative real parts, the operating point is stable; otherwise, the largest positive eigenvalue gives the rate of escape from the unstable operating point. Figure 5.6 shows the positive eigenvalue of the Jacobian of f for the unstable operating point. The effectiveness of the Schmitt trigger at escaping from the unstable operating point approaches zero as it gets closer to the ends of the hysteresis interval. Even at its peak, this eigenvalue is less than 2×10^{-4} for this example. In other words, this shows how quickly the probability of remaining in a metastable condition drops if, for example, the input to the Schmitt trigger makes a non-monotonic transition. Some designers have suggested that the hysteresis of a Schmitt trigger can be used to avoid metastability failures in synchronizer circuits. Thus, I compared the metastability of a Schmitt trigger with that of a two-inverter loop (the key subcircuit of a synchronizer flip-flop) (as shown in Figure 5.2 B). The two-inverter loop has a positive eigenvalue of 3×10^{-3} . Thus it is more than 15X

Figure 5.6: Eigenvalues of the Schmitt Triggers Unstable DC Operating Point

as effective at resolving metastability. These experiments used the short-channel MOSFET model for transistor currents and a “normalized” capacitance of 1.

Chapter 6

Conclusions

I presented a new approach for finding all DC equilibrium points of circuits using interval-arithmetic-based verification algorithms. While interval arithmetic has been used by solvers such as iSAT [9] and dReal [11], the use of interval verification algorithms (e.g. the Krawczyk method) for analog and mixed signal verification is new. I demonstrated our approach using a simplified, *tanh*-based model, a simple long-channel MOSFET model, and a state-of-the-art short-channel MOSFET model. Our solver can establish the existence and uniqueness of a solution within a given hyper-rectangle enabling our approach to find *all* solutions to a problem. I showed the first, fully automatic solution to the Rambus Ring Oscillator challenge problem [17]. Analysis of the equilibrium points and characterizing the metastable behaviour of a Schmitt trigger further shows the applications of our approach. Our approach showed much better scalability than the state-of-the-art solvers dReal [11] and Z3 [8].

Our solver is available as an open-source project in <https://github.com/iaakhter/nonLinearSolver>. I described and quantified implementation trade-offs that I encountered when developing this solver including the number of times the Krawczyk operator should be applied to a hyper-rectangle before resorting to bisection, the appropriate epsilon inflation to take care of solutions near the boundary and the bisection procedures that can be applied in our method.

Future work suggested by this research includes finding the right balance between using the Krawczyk operator and interval arithmetic so that the more expen-

sive Krawczyk operator is used only when necessary. The use of an incremental linear programming solver can also be explored. It might also be possible to incorporate an SMT or SAT solver in our method that would replace the linear programming part of our solver. Furthermore, extension of the Krawczyk method to solve underdetermined and overdetermined systems are also valid areas of research.

Bibliography

- [1] CVXOPT. <https://cvxopt.org/>. Accessed: 2018-11-03. → page 21
- [2] MC++. <https://omega-icl.github.io/mcpp/index.html>. Accessed: 2018-09-10. → page 26
- [3] G. Alefeld. Inclusion methods for systems of nonlinear equations-the interval newton method and modifications. *Topics in Validated Computations*, pages 7–26, 1994. URL <https://ci.nii.ac.jp/naid/10028169062/en/>. → page 8
- [4] C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation - using the forward and backward methods. Technical Report IMM-REP-1996-17, Department of Mathematical Modeling, Technical University of Denmark, 09 1996. → page 26
- [5] F. Benhamou and L. Granvilliers. Continuous and Interval Constraints. In P. v. B. F. Rossi and T. Walsh, editors, *Handbook of Constraint Programming*, pages 571–604. Elsevier, 2006. URL <https://hal.archives-ouvertes.fr/hal-00480814>. → page 5
- [6] O. Caprani and K. Madsen. Iterative methods for interval inclusion of fixed points. *BIT Numerical Mathematics*, 18(1):42–51, Mar 1978. ISSN 1572-9125. doi:10.1007/BF01947742. URL <https://doi.org/10.1007/BF01947742>. → page 16
- [7] G. E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proceedings 2nd GI Conference on Automata Theory and Formal Languages*, pages 134–183, Berlin, 1975. Springer. doi:10.1007/3-540-07407-4_17. URL http://dx.doi.org/10.1007/3-540-07407-4_17. LNCS 33. → page 5
- [8] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the*

Construction and Analysis of Systems, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3. → pages 1, 5, 31, 43

- [9] M. Fränzle. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 30(3):179–198, 2007. doi:10.1007/s10703-006-0031-0. → pages 6, 43
- [10] M. Fränzle, C. Herde, T. Teige, S. Ratschan, and T. Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007. → pages 1, 5, 6
- [11] S. Gao, S. Kong, and E. M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In M. P. Bonacina, editor, *Automated Deduction – CADE-24*, pages 208–214, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38574-2. → pages 1, 6, 31, 43
- [12] S. Gao, S. Kong, and E. M. Clarke. Satisfiability modulo ODEs. In *Proceedings of the 13th International Conference on Formal Methods in Computer Aided Design (FMCAD’13)*, pages 105–112, Oct. 2013. doi:10.1109/FMCAD.2013.6679398. URL <http://dx.doi.org/10.1109/FMCAD.2013.6679398>. → page 6
- [13] L. A. Glasser and D. W. Dobberpuhl. *The Design and Analysis of VLSI Circuits*. Addison-Wesley, 1985. → page 40
- [14] L. B. Goldgeisser and M. M. Green. A method for automatically finding multiple operating points in nonlinear circuits. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 52(4):776–784, April 2005. ISSN 1549-8328. doi:10.1109/TCSI.2005.844359. → pages 1, 4
- [15] M. R. Greenstreet and S. Yang. Verifying start-up conditions for a ring oscillator. In *Proceedings of the 18th Great Lakes Symposium on VLSI (GLSVLSI’08)*, pages 201–206, May 2008. doi:10.1145/1366110.1366160. URL <http://dx.doi.org/10.1145/1366110.1366160>. → pages v, 2
- [16] C.-W. Ho, A. Ruehli, and P. Brennan. The modified nodal approach to network analysis. *IEEE Transactions on Circuits and Systems*, 22(6): 504–509, June 1975. ISSN 0098-4094. doi:10.1109/TCS.1975.1084079. → page 26

- [17] K. D. Jones, J. Kim, and V. Konrad. Some “real world” problems in the analog and mixed-signal domains. In *Proc. Workshop on Designing Correct Circuits*, Apr. 2008. → pages 1, 31, 34, 43
- [18] D. Jovanović and L. de Moura. Solving non-linear arithmetic. In B. Gramlich, D. Miller, and U. Sattler, editors, *Automated Reasoning*, pages 339–354, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31365-3. → pages 1, 5
- [19] R. B. Kearfott. GlobSol: History, composition, and advice on use. In C. Blik, C. Jermann, and A. Neumaier, editors, *Global Optimization and Constraint Satisfaction*, pages 17–31, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-39901-8. → page 15
- [20] J. Kim, M. Jeeradit, B. Lim, and M. A. Horowitz. Leveraging designer’s intent: a path toward simpler analog CAD tools. In *Proceedings of the Custom Integrated Circuits Conference (CICC’2009)*, pages 613–620, Sept. 2009. doi:10.1109/CICC.2009.5280741. URL <http://dx.doi.org/10.1109/CICC.2009.5280741>. → page 1
- [21] R. Krawczyk. Newton-algorithmen zur bestimmung von nullstellen mit fehlerschranken. *Computing*, 4:187–201, 1969. → pages 1, 6, 9
- [22] J. Lee and H.-D. Chiang. Constructive homotopy methods for finding all or multiple DC operating points of nonlinear circuits and systems. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 48(1):35–50, Jan 2001. ISSN 1057-7122. doi:10.1109/81.903186. → pages 1, 4
- [23] M. S. Lundstrom and D. A. Antoniadis. Compact models and the physics of nanoscale FETs. *IEEE Transactions on Electron Devices*, 61(2):225–233, Feb 2014. ISSN 0018-9383. doi:10.1109/TED.2013.2283253. → pages 27, 29
- [24] R. C. Melville, L. Trajkovic, S. . Fang, and L. T. Watson. Artificial parameter homotopy methods for the DC operating point problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(6):861–877, June 1993. ISSN 0278-0070. doi:10.1109/43.229761. → pages 1, 4
- [25] M. Miller and F. Brewer. Formal verification of analog circuit parameters across variation utilizing SAT. In *2013 Design, Automation Test in Europe*

- Conference Exhibition (DATE)*, pages 1442–1447, March 2013.
doi:10.7873/DATE.2013.294. → pages 1, 5
- [26] R. Moore. A test for existence of solutions to nonlinear systems. *SIAM Journal on Numerical Analysis*, 14(4):611–615, 1977.
doi:10.1137/0714040. URL <https://doi.org/10.1137/0714040>. → pages 9, 24
- [27] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009. ISBN 0898716691, 9780898716696. → pages 1, 6, 9
- [28] T. Quarles. The SPICE3 implementation guide. Technical Report UCB/ERL M89/44, EECS Department, University of California, Berkeley, 1989. URL <http://www2.eecs.berkeley.edu/Pubs/TechRpts/1989/1218.html>. → page 3
- [29] S. Rakheja and D. Antoniadis. MVS nanotransistor model (silicon), Dec 2015. URL <https://nanohub.org/publications/15/4>. → page 40
- [30] S. Rump. INTLAB - INTerval LABoratory. In T. Csendes, editor, *Developments in Reliable Computing*, pages 77–104. Kluwer Academic Publishers, Dordrecht, 1999. <http://www.ti3.tu-harburg.de/rump/>. → page 7
- [31] S. M. Rump. Verification methods: Rigorous results using floating-point arithmetic. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 3–4, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0150-3.
doi:10.1145/1837934.1837937. URL <http://doi.acm.org/10.1145/1837934.1837937>. → pages 1, 2, 6, 16, 24
- [32] B. Schling. *The Boost C++ Libraries*. XML Press, 2011. ISBN 0982219199, 9780982219195. → page 26
- [33] S. Steinhorst, M. Peter, and L. Hedrich. State space exploration of analog circuits by visualized multi-parallel particle simulation. In *International Conference on Signal Processing Systems (ICSPS'09)*, pages 858–862, May 2009. → page 2
- [34] S. Tiwari, A. Gupta, et al. First steps towards SAT-based formal analog verification. In *Proceedings of the 2010 International Conference on Computer Aided Design*, Nov. 2010. → pages 1, 2, 5
- [35] D. Wolf and S. Sanders. Multi-parameter homotopy methods for finding DC operating points of nonlinear circuits. In *1993 IEEE International*

Symposium on Circuits and Systems, pages 2478–2481 vol.4, May 1993.
doi:10.1109/ISCAS.1993.394267. → pages 1, 4

- [36] M. H. Zaki, I. Mitchell, and M. R. Greenstreet. Towards a formal analysis of DC equilibria of analog designs. *Formal Verification for Analog Circuits (FAC'09)*, June 2009. → pages v, 6, 27, 28, 40

Appendix A

Appendix

A.1 C++ Short Channel MOSFET MODEL Code

```
/* Calculate ids from Vds, Vgs, Vbs according to the short channel MOSFET model.
 * @param params map of parameter variables to the subsequent values
 * @param Vds evaluation of (interval drain – interval source)
 * @param Vgs evaluation of (interval gate – interval source)
 * @param Vbs evaluation of (interval body – interval source)
 * @param iDir integer indicating the direction of current.
 *          iDir = 1 -> current from drain to source
 *          iDir = -1 -> current from source to drain
 * @return interval ids
 */
F<I> mvs_id(const std::map<std::string, double>& params,
            const F<I>& Vds, const F<I>& Vgs,
            const F<I>& Vbs, const int& iDir){
    std::string versionS = "version", mTypeS = "mType", WS = "W", LgdrS = "Lgdr";
    std::string dLgS = "dLg", CgS = "Cg";
    std::string etovS = "etov", delS = "delta", n0S = "n0", Rs0S = "Rs0";
    std::string Rd0S = "Rd0", CifS = "Cif", CofS = "Cof", vxoS = "vx0";
    std::string muS = "mu", betaS = "beta", TjunS = "Tjun";
    std::string phibS = "phib", gammaS = "gamma", Vt0S = "Vt0";
    std::string alphaS = "alpha", mcS = "mc";
    std::string CTM_selectS = "CTM_select", CCS = "CC", ndS = "nd", zetaS = "zeta";
    double version = params.at(versionS);
    double mType = params.at(mTypeS);
```

```

double W = params.at(Ws);
double Lgdr = params.at(LgdrS);
double dLg = params.at(dLgS);
double Cg = params.at(CgS);
double etov = params.at(etovS);
double delta = params.at(delS);
double n0 = params.at(n0S);
double Rs0 = params.at(Rs0S);
double Rd0 = params.at(Rd0S);
double Cif = params.at(CifS);
double Cof = params.at(CofS);
double vxo = params.at(vxoS)*1e+7;
double mu = params.at(muS);
double beta = params.at(betaS);
double Tjun = params.at(TjunS);
double phib = params.at(phibS);
double gamma = params.at(gammaS);
double Vt0 = params.at(Vt0S);
double alpha = params.at(alphaS);
double mc = params.at(mcS);
double CTM_select = params.at(CTM_selectS);
double CC = params.at(CCS);
double nd = params.at(ndS);
double zeta = params.at(zetaS);

// Virtual source not included here

F<I> Vdsi = Vds;
F<I> Vgsi = Vgs;
F<I> Vbsi = Vbs;

// F = Coulombs/V
// s-terminal outer fringing cap [F/cm]
double Cofs = 0*( 0.345e-12/ etov ) * dLg/ 2.0 + Cof;
// d-terminal outer fringing cap [F/cm]
double Cofd = 0*( 0.345e-12/ etov ) * dLg/ 2.0 + Cof;
// Effective channel length [cm]. After subtracting overlap lengths on s and d side
double Leff = Lgdr - dLg;

// Boltzmann constant [eV/K]

```

```

double kB = 8.617e-5;
// Thermal voltage ,  $kT/q$  [V]
double phit = kB*Tjun;
// Carrier mass [Kg]
double me = (9.1e-31) * mc;
// Total subthreshold swing factor taking punchthrough into account [unit-less]
F<I> n = n0 + nd * Vdsi;
F<I> nphit = n * phit;
double aphit = alpha * phit;

F<I> phibVbs = fabs(phib - Vbs);

F<I> Vtpcorr = Vt0 + gamma * (sqrt(phibVbs)- sqrt(phib))- Vdsi * delta;
F<I> eVgpre = exp(( Vgs - Vtpcorr )/ ( aphit * 1.5 ));
F<I> FFpre = 1.0/( 1.0 + eVgpre );
F<I> ab = 2 * ( 1 - 0.99 * FFpre ) * phit;
F<I> Vcorr = ( 1.0 + 2.0 * delta ) * ( ab/ 2.0 ) * ( exp( -Vdsi/ ab ));
F<I> Vgscorr = Vgsi + Vcorr;
F<I> Vbscorr = Vbsi + Vcorr;
F<I> phibVbscorr = fabs(phib - Vbscorr);

F<I> Vt0bs = Vt0 + gamma * (sqrt(phibVbscorr) - sqrt( phib ));

F<I> phibVbsi = fabs(phib - Vbsi);

F<I> Vt0bs0 = Vt0 + gamma * (sqrt( phibVbsi) - sqrt( phib ));

F<I> Vtp = Vt0bs - Vdsi * delta - 0.5 * aphit;
F<I> Vtp0 = Vt0bs0 - Vdsi * delta - 0.5 * aphit;
F<I> eVg = exp(( Vgscorr - Vtp )/ ( aphit ));
F<I> FF = 1.0/ ( 1.0 + eVg );
F<I> eVg0 = exp(( Vgsi - Vtp0 )/ ( aphit ));
F<I> FF0 = 1.0/ ( 1.0 + eVg0 );

//  $F/cm^2 \cdot V = Coulombs/cm^2$ 
F<I> Qref = Cg * nphit;
F<I> eta = ( Vgscorr - ( Vt0bs - Vdsi * delta - FF * aphit ))/ ( nphit );
// Using FF instead of FF0 in eta0 gives smoother capacitances.
F<I> eta0 = ( Vgsi - ( Vt0bs0 - Vdsi * delta - FF * aphit ))/ ( nphit );

```

```

F<I> Qinv_corr = Qref*log1p(exp(eta));

// Transport equations
double vx0 = vx0;
double Vdsats = vx0 * Leff/mu;
// Saturation drain voltage for current
F<I> Vdsat = Vdsats * ( 1.0 - FF ) + phit * FF;

F<I> VdsiVdsat = Vdsi/Vdsat;

F<I> powVal = pow(VdsiVdsat, beta);
// Transition function from linear to saturation.
// Fsat = 1 when Vds>>Vdsat; Fsat= Vds when Vds<<Vdsat
F<I> Fsat=(Vdsi/Vdsat)/(pow((1+powVal), (1.0/beta)));

// Total drain current
// Coulombs/s = A
F<I> Id = Qinv_corr * vx0 * Fsat * W * mType * iDir;
return Id;
}

```

A.2 Informal Proof of the Number of DC Operating Points in the Rambus Ring Oscillator with Strong Cross-coupled Inverters

Figure 5.3 shows a 2 stage Rambus ring oscillator where $V(0,0)$ and $V(1,0)$ constitute the first stage and $V(1,0)$ and $V(1,1)$ constitute the second stage.

Let's assume that a stage can be labeled with either H , L or M . In the example in Figure 5.3, if the first stage is at H , $V(0,0)$ is at a high value and $V(0,1)$ is at a low value. Likewise, if the first stage is at L , $V(0,0)$ is at a low value and $V(0,1)$ is at a high value. Finally, if the first stage is at M , both $V(0,0)$ and $V(0,1)$ are at mid value. High voltage holds a value close to the power; low voltage holds a value close to the ground; middle voltage holds a value close to the mid point between low and high. A voltage assignment across n stages can be represented by a string of length n consisting of H , L or M characters. For example, for a two

stage oscillator a string HL can represent a voltage assignment. This means that the first stage has an H assignment and the second stage has an L assignment.

Assume that we remove the forward inverters. Each of the cross coupled inverters independently has 2 DC operating points. Thus for n stages there are 2^n operating points. When cross coupled inverters are very strong when compared to the forward inverters, for example when $gcc = 4.0$, for an n stage oscillator, there would still be 2^n stable operating points — all possible combinations of H and L for each stage.

Now consider any assignment for an n -stage oscillator consisting of characters H , L or M that represents a DC operating point. In such an assignment, we can turn any character that is not M into an M and that assignment will still be a DC operating point. Suppose for a six stage oscillator we have the assignment $HLLHLH$ and we turn it into $HMLHLH$ by changing the voltage assignment to the second stage accordingly from L to M . The change in the voltages at the second stage will be propagated across to the third stage and the forth stage and so on and then back to the second stage. However, at each stage forward the change will decrease and would not affect the existing assignment of the stages (either H , L or M) because of the weak forward inverters until eventually the voltage assignments will settle to a fixed point which would be the DC operating point.

Thus, we first showed that an n -stage Rambus oscillator has 2^n stable DC operating points indicated by any combination of H or L for each stage. Then we showed that any character in an existing DC operating assignment that is not M can be turned into one and that assignment would still represent a DC operating point. This means that for an n stage oscillator, each stage can have either H , L or M assignment at a DC operating point. Hence there are 3^n DC operating points.