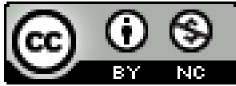
FIXTURE: A TOOL FOR AUTOMATED MODELING OF MIXED-SIGNAL SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Daniel Stanley
September 2023

© 2023 by Daniel Stanley. All Rights Reserved.
Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.
http://creativecommons.org/licenses/by-nc/3.0/us/

This dissertation is online at: https://purl.stanford.edu/qg678dn6395

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Boris Murmann

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Priyanka Raina

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format.

Abstract

Functional modeling of analog circuits is an important step in the verification of today's mixed-signal Systems-On-Chip, and it will only become more important as analog-digital interaction becomes more common. The Verilog functional models must be written by a skilled engineer, but there is significant repeated code between models for similar circuits. Some tools have been created to automate this repeated work including the DaVE toolset which uses a library of templates to organize analysis and modeling strategies for different types of analog circuit, automating the entire spice-to-Verilog flow for many circuits. This thesis seeks to extend the functionality of the DaVE toolset and create a new open-source analog circuit analysis tool called Fixture. Fixture is able to characterize a wider variety of user circuits than DaVE, and users can combine these new analysis templates with the existing model templates in DaVE to automate the flow from spice netlist to Verilog functional model.

The features we added with Fixture were motivated by our attempts to model real-world circuits using DaVE and the issues we encountered with certain user circuits. Rather than make individual fixes for each circuit we worked to generalize the fixes so they could apply to a wide variety of user circuits and a wide variety of circuit types across the template library. This led to improvements with modeling nonlinear circuits, managing model complexity, and debugging user inputs, among many others. In addition to making user-facing improvements, we also gave Fixture a modular design so that any user can contribute to the open-source repository with new templates and tests.

This thesis will walk through the features of Fixture and explain both how and why Fixture operates the way it does, covering many aspects of the tool. We make the process of template creation easier for engineers by organizing templates into tests. For each test, the engineer only needs to write the details specific to that test while Fixture automates general tasks like choosing sample points, performing regression, and plotting results. We use the *fault* library to allow templates to have a single testbench for both spice and Verilog circuits. Additionally, this allows Fixture to augment handwritten testbenches to add stimuli for additional inputs that modify circuit behavior, change the domain for an input or output signal, or convert a testbench for a single-ended circuit into one for a differential circuit. After collecting data, Fixture automatically solves a nonlinear optimization problem to fit coefficients in an equation provided by the template writer. The template writer can

also supply multiple equations to give the user freedom in the tradeoff between accuracy and speed in the final model. In addition, the owner of the circuit being modeled has precise control over the way various inputs affect parameters of the equation, including the ability to specify arbitrary nonlinear relationships with their own coefficients to be fit to the data. Fixture intelligently chooses sample points to use in circuit simulation to accurately fit these coefficients while reducing simulation time. Finally, Fixture uses these same sample points to produce plots of various circuit parameters to allow engineers to quickly verify circuit performance or debug any issues. These improvements over previous automated modeling tools have allowed us to create models for real-world circuits that could previously only be modeled by hand. We hope that as new engineers use the tool, the library will become more robust and more useful.

Fixture can be found at `https://github.com/standanley/fixture`.

Acknowledgements

First and foremost I would like to thank my advisor, Mark Horowitz, for the PhD experience you have given me. I appreciate the amount of time you invest in each of your students, and thanks for helping me with everything from career advice to tiny bugs in my code. I really enjoyed working with you.

Thank you to the other students in the circuits group - Sung-Jin, Steven, Zach, Can, Sunil, Luke - you were my best way of getting unstuck, and I learned a ton from chatting with all of you about our various projects or just EE in general. Also thank you to Byong for essentially handing off your project to me; I'm grateful to have had such a solid base to build from. To the rest of Mark's group, thank you for group lunch feedback, fun on the group trips, making dinner and playing video games together, ski trips, and everything else.

Thank you to my friends - on the West Coast, the East Coast, and virtual - for board games, D&D, playing in the snow, art nights, and lots more. The pandemic made a huge mess of things, and I'm grateful to Teresa, Jacob, and Lucy for figuring out how to make some good out of it and for keeping me moving on my PhD at the same time. Thank you especially to my girlfriend Lucy for always staying supportive and keeping me motivated, I couldn't have done it without you.

Finally, thank you to my parents for everything you've done to help me get here. I'm grateful that you've given me nothing but support throughout this whole journey, and I truly wouldn't be here without your help.

Contents

Abstract **iv**

Acknowledgements **vi**

1 Introduction **1**

2 Prior Work **4**

2.1 Analog / Mixed-Signal Simulation 4

2.2 Scientific Analog 5

2.2.1 xmodel 5

2.2.2 modelzen 6

2.2.3 glister 7

2.3 Other Spice Model Verification Techniques 7

2.4 DaVE Tools 9

2.4.1 DaVE: A Template Library 9

2.4.2 A DaVE Template 10

2.4.3 The DaVE Environment 14

2.4.4 An Evolving Library 16

3 A Motivating Circuit **19**

3.1 The Challenging Circuit 19

3.1.1 User input 20

3.1.2 Choosing Sample Points 22

3.1.3 Simulation 23

3.1.4 Model fitting 23

3.2 Guiding Principles 25

3.2.1 Provide Reasonable Defaults, but Advanced Options 25

3.2.2 Follow a Process that is Familiar to Engineers 26

3.2.3 Make Data User-Accessible 27

4 A Template Library **28**
4.1 Structure of a Fixture Template 28
4.2 Parameter Equations 31
4.3 Optional Input Types 34
4.3.1 Pinned Values 34
4.3.2 Analog 34
4.3.3 Quantized Analog 35
4.3.4 True digital 37
4.3.5 Load Specification 37
4.3.6 Challenges with Process / Temperature variation 38
4.4 Updating Equations with Optional Inputs 39
4.4.1 Equation Hierarchy 39
4.4.2 User Configuration: Optional Input Dependence 41
4.5 Extending a Template with Vectored Inputs and Outputs 43
4.6 Extending a Template with Custom Domain Changes 46
4.6.1 Linear Transformations 47
4.6.2 Time-Based Transformations 48
4.7 Using the Template 50

5 Testbench Generation **51**
5.1 Choosing Input Points 51
5.1.1 Latin Hypercube Sampling and Orthogonal Sampling 52
5.1.2 Scaling Input Samples 52
5.1.3 Custom Input Constraints 54
5.2 The Testbench Description Language *fault* 55
5.2.1 Testbenches Written in Python 56
5.2.2 Domain Translation 57
5.3 Templatized Testbenches 62
5.3.1 Writing a Templatized Testbench 62
5.3.2 Vectoring a Testbench 63
5.3.3 Optional Input Timing 65

6 Model Fitting **66**
6.1 Regression 66
6.1.1 A Challenging Example 66
6.1.2 Challenges with Nonlinear Fitting 67
6.1.3 An Unsuitable Sampling Approach 69
6.1.4 Fixture's Improved Approach 72

6.1.5 Additional Nonlinear Fitting Techniques 74
6.2 Plotting 76
6.2.1 Fixed Optional Input Plots 76
6.2.2 Parameter vs. Optional Input 79
6.2.3 Final Model 81
6.2.4 Contour Plots 81
6.3 Additional Outputs from Fixture 83
6.4 Fixture Checkpoints 86
6.5 Preliminary Model Generation 86

7 Conclusion **88**

List of Figures

2.1 Phase Blender Explanation 11
2.2 Phase Blender Smooth Relationship 12
2.3 Dave Environment 15
2.4 Phase Blender Glitching Behavior 18
2.5 Phase Blender Wrapping 18

3.1 Challenging TIA 20

4.1 Template Structure 29
4.2 Amplifier Nonlinearity Comparison 33
4.3 Example Wrapper for Circuit with Dynamic Loading and Process Variation 38
4.4 Vectoring Dynamic Response Ambiguity 45
4.5 Custom Domain Translation 47
4.6 DragonPHY Sample-and-Hold Timing 49

5.1 Input Sample Distribution 53
5.2 Histograms Showing Issues with Independent Bit Sampling 54
5.3 Time Domain Read 61

6.1 Basic Linear Model Scatterplot 70
6.2 Estimating Parameter from Incomplete Data 71
6.3 Fixture's Sampling Strategy 73
6.4 Fixed Optional Input Plots 77
6.5 Fixed Optional Input Plots - Debugging Example 78
6.6 Parameters vs. Optional Inputs: radj 79
6.7 Parameters vs. Optional Inputs: vdd 80
6.8 Reciprocal Init Trick Debug 81
6.9 Final Model Plots 82
6.10 Final Model Plots vs. Optional Inputs 83
6.11 Contour Plots - Successful Example 84

6.12 Contour Plots - Debugging Example 85

Chapter 1

Introduction

For decades, device engineers have been shrinking transistors for improved speed and low-power performance. This has worked wonders for digital circuits, resulting in the incredible computing power we have today. Unfortunately, the benefits that digital circuits get from this scaling do not always apply to analog circuits as well. In fact, optimizations made by device engineers to benefit digital circuits can have detrimental effects when used in analog designs [1]. If we want improved digital circuits we cannot leave analog behind: more processing power will come with a need for faster wired and wireless communications, both of which depend on improved analog performance. Even fully-digital chips always include some analog components for power management and for communicating data on and off chip. We need some way to apply transistor scaling improvements to analog circuits.

Besides relying on improved analog devices directly, there are several ways we can use improved digital devices to compensate for the limited performance of analog circuits [2, 3]. First, we can simply use more sophisticated digital algorithms to compensate for limited analog performance [4–6]. For example, in a high-speed link receiver the analog Continuous-Time Linear Equalizer (CTLE) or even the sampling phase adjustment can be eliminated and replaced with digital processing [7]. Second, we can improve analog circuit performance through the use of digital trim bits [8]. Imperfections in analog circuits due to process variation can be detected and corrected as part of a calibration routine. For example, the linearity of a time-interleaved Analog-to-Digital Converter (ADC) can be improved by measuring and correcting the offset in each individual channel. By estimating the offset statistically over many measurements and applying the correction in the analog domain, the correction can have sub-Least-Significant-Bit (LSB) precision, which is better than what is possible with digital post-processing alone [9].

Methods that measure an imperfection and apply a fix to the analog circuitry, rather than simply making a numerical correction in the digital domain, can achieve better performance at the cost of greater coupling between the analog and digital blocks. This greater coupling leads to more

effort in design and verification, and a higher likelihood of mistakes in the circuit [10, 11]. To make matters worse, this complexity often comes in the form of a feedback loop between analog and digital circuitry, meaning that the design process is often split between an analog and a digital circuit design team. The interface between analog and digital blocks is at high risk for design mistakes because it relies on communication between different engineers on different teams who work at different levels of design abstractions to agree on a specification [12]. Verifying the interface through simulation is also challenging because digital calibration and adaptation loops can take many cycles to verify, but analog circuits take significant computation to simulate each cycle. Chip design teams address this challenge through a combination of block-level testing, interface specifications, mixed-signal simulations, and tests using functional models of the analog circuits.

Functional modeling presents an interesting opportunity because it opens the door to system-level testing while creating new challenges in the verification of the functional model itself. The idea behind functional modeling is to write a Verilog[1] module which has the same behavior as an analog block. This model can then be used in place of the spice representation of that block for simulation, turning a mixed-signal simulation into a fully-digital simulation. With carefully-written analog models, these fully-digital simulations are significantly faster than their mixed-signal counterparts. This allows engineers to simulate the hundred thousand or more cycles necessary to validate the feedback loops between analog and digital components. In extreme cases, the models can even be optimized for an FPGA in order to run the trillions of cycles necessary for Bit Error Rate (BER) testing [13]. Besides simulating more cycles, functional models also allow engineers to simulate more blocks at once, enabling an entire SoC to be simulated at once [14].

Engineering teams have settled on functional model creation as a necessary piece of modern SoC verification, but the creation of functional models is still a challenging and specialized process. Knowledge of analog circuit behavior and digital simulation techniques are both necessary to create a good model, making it relatively difficult to find engineers suited for the task. These engineers must work with the analog design team to understand the important nonidealities of the circuit being modeled and ensure that the correct behaviors are represented in the model. They must also work with the digital verification team to understand the different testbenches the model will be used in to strike the right balance of speed and performance of the model. While this makes it challenging to write a new functional model from scratch, it is fortunate that many of the models being written overlap significantly with previous models. Designs that are updates to a previous generation or ports to a new process node are often similar enough to a previous model that the only changes are updates to coefficients or bus widths. We believe that this combination of a highly specialized engineer doing largely repetitive work gives ample motivation and opportunity to automate functional model generation.

Ideally, we desire a tool that takes a spice model as input and produces a fast and accurate

[1] Throughout this thesis, "Verilog" will be used to refer to both Verilog and SystemVerilog.

Verilog functional model as output. Of course, this is already a challenging problem for a human engineer, and an automated tool that works in all situations is out of reach for now. Still, many teams have already created automated tools that tackle this problem in specific contexts [15] (focusing on a subset of circuits), or focus on certain subproblems [16–19] (verifying existing functional models, help engineer to hand-write models), or address the same problem in a slightly different way [20] (using a custom non-Verilog simulator). Each of these approaches has its own advantages and disadvantages which we will discuss in Chapter 2. The world of digital design tools is more mature than that of analog tools [21], but we believe that the complexity of analog design should not prevent us from having equally capable tools for analog and mixed signal design.

This thesis describes my work in taking an existing approach to automated functional modeling, the DaVE ecosystem, and making significant changes and improvements resulting in the tool Fixture. Section 2.4 describes the state of the DaVE ecosystem at the start of the Fixture project, which includes an explanation of the template-library approach that both DaVE and Fixture take to modeling. Chapter 3 gives an example of a circuit that DaVE could not model well, serving as motivation new functionality in Fixture. The next three chapters describe Fixture itself, discussing in detail the solutions to the challenging problems presented in Chapter 3. Chapter 4 focuses on the template library system. It describes the work that a template writer needs to do to add a new circuit type to the Fixture library, as well as the steps the tool can perform automatically to extend a template to match a particular user's instance of a circuit. Chapter 5 explains how Fixture generates a testbench, including the choice of sample points, compilation to different simulators, and the hand-writing and automatic extension of templatized testbenches. Next, Chapter 6 describes the model-fitting process. This chapter explains the many challenges and solutions encountered when replacing DaVE's existing linear model fitting to nonlinear models, with a special focus on how to help the user understand and debug this more complicated process. Finally, Chapter 7 concludes the thesis and offers some insight into future research directions.

Chapter 2

Prior Work

Engineers have tackled the problem of mixed-signal modeling with many different approaches. The work in this thesis, Fixture, has the capability to automatically characterize a wide variety of circuits. This characterization is mainly used to produce functional models, but can also be used to verify existing functional models. In order to give points of comparison for each of these features, this section will discuss tools that can model specific circuit types, model multiple circuit types, verify that existing testbenches cover all the circuit behaviors, and verify that existing functional models match the circuit. Additionally, we will describe prior work on the DaVE set of tools since Fixture takes the same approach to circuit modeling and ultimately fits in as another tool in the DaVE ecosystem.

2.1 Analog / Mixed-Signal Simulation

Before discussing various approaches to analog functional modeling, we will first discuss Analog / Mixed-Signal (AMS) simulation as a baseline. The main advantage of AMS simulators is that they allow for designers to directly use the circuit descriptions they already have in the analog and digital domains, eliminating any additional modeling work and any mistakes that could arise during that work. Unfortunately, AMS simulation is often not fast enough to handle large system-level or long-running testbenches.

Because many circuit design teams already use Cadence Spectre [22] for circuit simulation, Cadence Spectre AMS is a popular mixed-signal simulator [23]. It combines two existing commercial tools, inheriting the benefits of highly-tested and well-known simulators on both the analog and digital sides. There is significant benefit in using an AMS simulator that is compatible with the technology models and simulator settings already being used by the analog design team, and this makes it hard for many teams to switch to any other AMS simulator.

An alternative to Cadence Spectre AMS is SystemC AMS. SystemC AMS offers more flexibility

in the implementation of analog blocks, including control over the way timed signals are abstracted to reduce events [24, 25]. Additionally, SystemC AMS is transparent about the way the internal circuit solver runs. In some cases this can allow users to change their circuit representation to optimize simulation speed without changing the results. One user was able to optimize their model of a resistive crossbar array based on the specific matrix operations being done by the SystemC AMS simulator, reducing simulation time by 93% compared to Cadence Spectre [26]. In Section 2.3 we also see SystemC AMS models used as an analog representation language for model comparison.

Other mixed-signal simulators exist with different capabilities. For example, FIDELDO [20] is a simulator that significantly predates both Cadence Spectre AMS and SystemC AMS. It is notable because, in addition to high-level digital descriptions and spice-level analog descriptions, it allows for functional analog descriptions in the Laplace-domain and z-domain. Laplace domain modeling will be discussed in the next section in the context of a more modern tool, *xmodel* [27].

2.2 Scientific Analog

Scientific Analog is a company with its own set of tools for automated functional model generation. *xmodel* provides a library of SystemVerilog building blocks that model circuit behavior in the Laplace domain [27]. *modelzen* can automatically convert spice netlists to *xmodel* [28]. Finally, *glister* provides a graphical interface to the previous two tools for engineers working in Cadence Virtuoso [29].

2.2.1 xmodel

xmodel is organized as a library of individual building blocks that can be used to construct analog circuit models, but each block takes a Laplace-domain approach to modeling an input-output relationship. Every signal in *xmodel* is represented using a datatype called XREAL. Rather than representing a single voltage, as is normally the case for SystemVerilog's real type, each XREAL variable holds a set of Laplace coefficients. Those coefficients correspond to the following waveform as a function of time:

$$\sum_i c_i \cdot t^{m_i-1} e^{-a_i t} u(t) \tag{2.1}$$

The same waveform can be represented in the Laplace domain as a function of s:

$$\sum_i \frac{b_i}{(s+a_i)^{m_i}} \tag{2.2}$$

Notice that the number of coefficients needed to represent this variable depends on i; indeed, the SystemVerilog implementation of this datatype links to a C variable whose size can vary.

The advantage of using this datatype is that it can represent the behavior of a linear circuit with

a single set of coefficients. In other words, as long as the input is not changing and the circuit is linear, the coefficients do not need to be recalculated. Additionally, cascading multiple linear circuits will always result in an output waveform that is well-represented by this datatype. Even in circuits with a strong non-linear element, this type of analysis can be beneficial. For example, many high-speed links use a Decision Feedback Equalizer (DFE) receiver, which is made up of a pre-emphasis filter, channel, linear equalizer, sampler, and DFE filter. Of these, every block is well-modeled as linear and time-invariant (LTI) except the sampler. Luckily, the output of the sampler only needs to be updated once per clock cycle so the number of coefficient updates that *xmodel* needs to do is small. As a result the entire system can be modeled with a small, constant number of events per clock cycle, resulting in fast simulation times [30].

The drawback of *xmodel* is apparent when the circuit being modeled is not linear. In the case of the sampler we are able to make an optimization based on the limited output update rate, but in general *xmodel* uses a piecewise-linear approximation of the circuit's transfer function. In this case, an event must occur to update the coefficients each time the circuit crosses from one linear piece to another. Additionally, the tool needs to determine when these crossings will occur, which requires additional computation. When the number of piecewise linear sections is small, *xmodel* can still out-perform time-domain representations such as those in a traditional spice simulator or other functional models [31]. This approach, however, does not scale as the number of piecewise linear sections increases, which is required if the circuit is strongly nonlinear or even if it is weakly nonlinear but the required accuracy is high.

It is worth noting that there are other ways to extend the Laplace representation to nonlinear circuits besides breaking the behavior into linear pieces. The creators of *xmodel* have also published results on using a Volterra Series approximation of weakly nonlinear circuits, and have found improved performance in some cases [32].

2.2.2 modelzen

modelzen is able to automatically build *xmodel* models of a user circuit by analyzing the netlist and running simulations. The first step is to break the user's circuit into chunks that can be modeled independently. Because of the event-based nature of digital simulation, functional models are always feed-forward, so any portion of a circuit with continuous-time feedback cannot be split into multiple blocks. The wire driving a transistor gate can typically be treated as strictly feed-forward, so *modelzen* uses these connections as the breaks between independently-modeled chunks. The tool automatically inspects the circuit netlist to find these breaks and group the remaining channel-connected components [33]. Component parameters are automatically parsed from the spice netlist and inserted into the appropriate *xmodel* blocks. The behavior of each component can also be determined through simulation and modeled as piecewise-linear in the Laplace domain using *xmodel* primitives. During the hand-modeling of a large DRAM array, *modelzen* was able to handle the

model creation for the 9-transistor bit-line sense amplifiers [34].

Although *modelzen* can save significant engineering effort by creating models automatically, it is limited in the types of circuits it can model well. Unlike *modelzen*, an engineer who is familiar with a particular analog circuit can consider the larger context of how each channel-connected component behaves and make intelligent decisions about how to appropriately simplify, combine, or eliminate them in the model. As a result, an engineer can manually use *xmodel* primitives to describe the high-level behavior of a circuit and create more efficient models than *modelzen* for some circuits. We will discuss in Section 2.4 how DaVE, and ultimately our tool Fixture, aim to codify the strategies used by engineers for specific circuit types to automatically apply them to future circuits.

2.2.3 glister

Finally, glister is a graphical interface to both *xmodel* and *modelzen*. Although it does not add functionality in terms of circuit modeling, it makes it easier for users to create models and interpret results. Automated generation tools are only useful if users are able to take advantage of them, so the importance of a good user interface should not be underestimated.

2.3 Other Spice Model Verification Techniques

Although this thesis focuses mainly on functional model generation, there are other techniques for verifying spice model correctness. In this section we will consider examples of spice testbench coverage metrics, comparison between spice model and high-level functional description, and comparison between spice and behavioral models. Although the approach is very different, these tools and our tool Fixture are ultimately trying to achieve the same goal: to verify the correctness of the user's circuit. This means that the issues addressed by these tools are likely issues that Fixture needs to address as well. For example, some tools aim to verify that existing testbenches exercise all important circuit behavior; we will also need to check whether Fixture's testbenches achieve this goal (Sections 2.4.2, 5.1). Other tools check whether an existing functional model matches the circuit; we should also have a strategy for analyzing Fixture's produced models (Section 5.2). In short, we can use this previous work to be better critics of our own work and ensure we are not ignoring any aspects of the problem.

With or without functional modeling, analog design engineers are expected to write testbenches for the analog blocks they create. In the world of software development there are code-coverage metrics to ensure that hand-written tests are accurately covering all aspects of the program being tested, but no standard exists for doing something similar for analog circuit tests. Several coverage metrics for analog circuit testbenches have been proposed to fill this gap. Sanyal et al. [16] propose a technique for measuring coverage by checking the range of values reached by each signal in an analog testbench, with special consideration for analog-specific features such as signal slope and glitches.

The authors then extend this concept to include cross-coverage, meaning that the testbenches can be required to exercise combinations of two different metrics simultaneously [17]. This is useful for finding behaviors that only occur when multiple signals are at their extreme value simultaneously. We apply a similar concept in our tool Fixture to generate sample points that cover all circuit behaviors (Section 5.1).

Fürtig et al. propose a similar coverage metric [18], applying a numerical measure to the percentage of the state space covered. For small circuits, however, these authors are also able to apply a different approach: the set of states covered over time from an initial starting set of states can be computed using a hybrid automaton. This same technique is used and explained by Lee et al. [35]. In short, the space of all possible states for a nonlinear analog circuit is discretized, and a set of states is defined as a set of discrete states and limits on continuous variables within those states. A transient simulation can then be run on sets of states simultaneously by using a linearized model of each discrete state to determine the dynamics. This is reminiscent of formal verification methods for digital circuits which can reason about sets of states simultaneously [36]. The main challenges for applying these concepts to analog designs are the accuracy and the computation time; generally accuracy is increased at the cost of more computation time by discretizing into smaller pieces. So far, this has limited the strategy to small designs with around a dozen state variables. Additionally, this strategy is only useful when the set of reachable states is bounded or converges to a stable point after some time. This makes it useful, for example, for testing the startup behavior of a bias generator or the settling behavior of a dynamic amplifier with a fixed input but prevents it from being used to verify circuits with changing output such as an oscillator.

Next, we will consider several tools doing model comparison. These tools start with a spice model of a circuit and a pre-written functional model of the same circuit, and check whether the behavior of the two is equivalent. One way to approach this problem is direct waveform comparison. Both models are simulated with the same input stimulus, and their outputs are recorded and compared. Most often, the output waveforms are in the voltage-vs.-time domain and deviation between the waveforms is quantified as a difference in voltage and/or time. It is up to the engineer to determine whether a particular amount of deviation is within an acceptable range or not. Cadence amsDmv has support for this type of waveform comparison, plus additional features like glitch detection and glitch removal [37]. In many cases, it is better to consider circuit inputs and outputs in a domain other than the voltage-vs.-time domain (see Section 2.4.2 for further discussion and examples). For example, Coskun et al. [15] created a model checker than only applies to linear filters, and as a result is able to work in the Laplace domain. Their tool compares the transfer function of a spice netlist to a functional model written in SystemC AMS.

Finally, the idea of model checking in alternative domains is extended to cover many circuits by Singh et al. [38]. In their work, the domain of each input and output of a circuit is specified, and possible domains include frequency, phase, and charge. The tool automatically generates random

testbenches working in the specified domains and runs them on both the spice and functional models. The outputs are compared in the proper domain to see if they match within a specified error tolerance. Additionally, the tool can automatically search the input space to find the worst-case deviation between the two models. This approach is very similar to the model-checking strategies of DaVE (Section 2.4) and Fixture. The difference is that DaVE and Fixture model the collected results with a set of parameters first, and then compare the parameters. Both DaVE and Fixture have other capabilities as well, and DaVE will be discussed in the next section.

2.4 DaVE Tools

The Big-Digital, little-analog Verification Environment (DaVE) tools follow a template library approach to automatically model analog blocks [39, 40]. We will discuss DaVE in detail both as a review of prior work and because the creation of Fixture was motivated by DaVE tools. We will explain how DaVE works (Section 2.4), present some of the challenges that DaVE could not handle (Chapter 3), and then describe how Fixture overcomes those challenges (Chapters 4, 5, 6).

2.4.1 DaVE: A Template Library

When comparing automated modeling frameworks we find it helpful to place them on a spectrum from general to specific. General frameworks use one approach to model many different circuits, while specific frameworks use a different approach for each circuit. We can use this lens to evaluate a template-based modeling framework and discuss the advantages and disadvantages of DaVE.

One can imagine a tool that uses a single modeling strategy for every type of circuit. For example, the tool could use a Finite Impulse Response (FIR) filter and simply adjust the parameters of the filter to match the behavior of the given circuit. The biggest benefit of this one-size-fits-all approach is that the tool creator can spend a relatively small amount of time on the tool itself compared to the number of useful circuit models it can produce. In other words, extending one model to many different circuits saves a lot of engineering effort. At the same time, it makes using this modeling strategy harder for each user. While our FIR model can be made more accurate by reducing the timestep, and can even handle nonlinear circuits by dynamically adjusting the filter coefficients based on the circuit state, it puts the burden on the user to figure out how and when to adjust these parameters appropriately. These models also have a fixed trade-off between accuracy and performance which might not match the modeling goal. This limited approach to modeling also means there will always be circuits or specific circuit behaviors that the tool cannot model efficiently or cannot model at all. How would an FIR filter be used to model an output glitch that only appears when the relative timing of two inputs is in a certain range (for example, the phase blender glitching behavior in Figure 2.4)? The most general modeling strategy, using one approach for every circuit, saves a lot of engineering effort but is ultimately limited in the number of circuits it can model well.

On the other end of the spectrum, one can imagine specifically hand-modeling every circuit individually rather than building an automated tool. In this case every possible circuit is well-modeled, or at least modeled as well as the state of the art. Although this approach covers every circuit, and can employ any trick or strategy to create the best model, it saves no engineering effort through the use of automation.

DaVE uses a template library to try to capture the benefits of both the general and specific modeling approaches in one framework. DaVE includes a collection of templates where each template has instructions specific to modeling one type of circuit. For example, there is one template for all amplifiers, another for all sample-and-hold circuits, and another for all phase blenders. Then, the tool has the ability to modify and extend those templates to match the user's specific instance of a circuit. Extensions might include the addition of a bias current or calibration inputs. The extensions allow each template to apply to a variety of user circuits [41].

The goal is to have a large library of basic templates and to combine it with a long list of extensions that can be made to any template. Unfortunately these two goals are sometimes at odds, since it can be difficult to write an extension that works with a wide variety of circuit templates and difficult to write a circuit template that is compatible with every possible extension. DaVE addresses this problem by choosing a central abstraction that every template must conform to. This way the tool creator can write extensions that are compatible with the central abstraction without having to consider every template in the library. The downside to this approach is that it provides a constraint on the template design which may make it more difficult to write templates for certain types of circuit. In choosing the central abstraction there is some tradeoff between giving flexibility to the model writer and giving flexibility to the extensions. The specifics of the central abstraction used by DaVE will be discussed in the next section.

2.4.2 A DaVE Template

Although templates contain circuit-specific details, the ability to extend a template to handle different user circuits is a universal strategy that the tool can apply to any template. This need for universal modifications restricts the templates to have specific properties that the tool can modify in a universal way. With DaVE, each template must follow a specific format: the inputs and outputs are transformed into a space where they follow a linear, or piecewise linear, relationship. We will call this relationship the "central relationship" of a circuit. For a differential amplifier, this means the positive and negative input voltages are transformed into a differential and common-mode voltage, and the template works in the differential / common mode space. As another example, we will consider a phase blender circuit (Figure 2.1). For a phase blender, signals are transformed from the voltage vs. time domain to the phase domain. Before this transformation the output is a digital signal, so the output value changes abruptly at each edge. After the transformation the output surface is smooth, as demonstrated in Figure 2.2. This notion of transforming each circuit to its

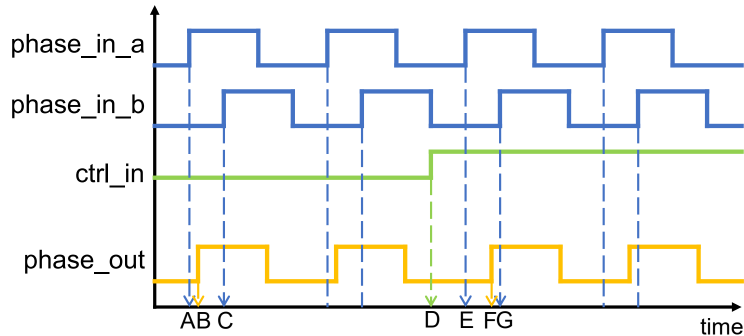


Figure 2.1: Diagram of phase blender operation. A phase blender takes two input clocks with different phases (`phase_in_a` and `phase_in_b`) and produces an output clock (`phase_out`) whose phase is somewhere in between. Phase blenders typically have an additional input (`ctrl_in`) to adjust the exact position of the output phase. In this example, when `ctrl_in` is low we can see that the output phase is closer to the phase of `phase_in_a` than `phase_in_b` (referring to the labels on the time axis, B is closer to A than to C). When the value of `ctrl_in` increases (D), the output edge moves later (F is closer to G than to E). Typically the output edge will occur at a time in between the two input edges, but there can also be delays due to buffers that move the output edge a constant amount of time later.

inherently "analog" domain works well with a template library: each template can correspond to a different domain transformation.

The fact that the central relationship is smooth in the proper domain not only makes modeling easier, but also allows DaVE to effectively capture circuit behavior through simulation. Because simulation time is finite it is impossible to test every possible stimulus for a continuous-value analog input. This means the functional model will sometimes have to guess an output for an input that DaVE has never simulated before. If we make the assumption that the circuit response is smooth, predicting the output for a never-before-seen input can be done by interpolating between nearby known input-output pairs. In other words, a non-smooth response may have spikes or discontinuities that would not be visible to the modeler unless the corresponding inputs were sampled directly; we rely on the smoothness assumption to guarantee that our model is not missing any behaviors. We consider this assumption to be the definition of an analog circuit: if the response is not smooth, then the circuit is not analog and DaVE makes no claim that it is able to model the circuit. In practice, essentially all real-world analog circuits conform to this smoothness assumption and could be modeled by DaVE with the proper template [42, 43].

Because of the smoothness assumption DaVE is able to characterize circuits with simulation results only, which has the additional benefit that DaVE never needs to consider the implementation of the circuit. By ignoring the implementation, DaVE is able to use one template for all implementations of a particular type of circuit. For example, amplifiers can be implemented with a single transistor, or in a cascode configuration, or as multiple gain stages in feedback, but the

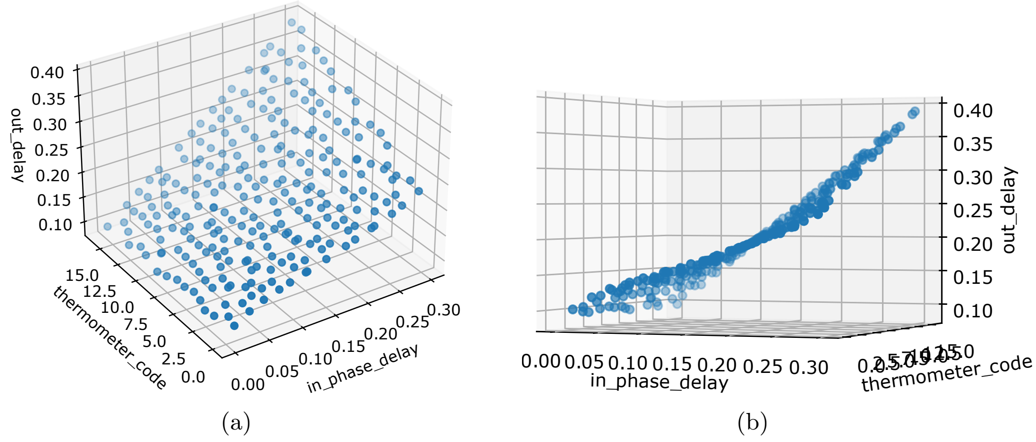


Figure 2.2: Two different view of the same simulated input-output pairs from a phase blender. (a) Note that the `in_phase_delay` and `out_phase_delay` axes are in the phase domain, which allows the response to be smooth. (b) Although the response is smooth, it is not linear: the results do not lie on a flat plane. Although this may present a challenge for modeling the phase blender, it does not affect our ability to capture the entire response with a finite number of sample points.

same testbench can characterize all of these. Additionally, the same testbench stimulus can be used to exercise a functional model of the circuit to verify that it has the same behavior as the original circuit.

DaVE is able to extend each circuit template by adding additional inputs to the circuit and allowing them to modify the central relationship [44]. The inputs that go through domain transformations and through the central relationship as described in the previous paragraph are called "required" inputs. Any additional inputs that modify the relationship are called "optional" inputs. For example, for a phase blender circuit the "required" inputs are the A and B input clocks to blend between (Figure 2.1). Every phase blender will have these two inputs, and they need to go through a transformation from the voltage vs. time domain to the phase domain. Although most blenders also have a control input to adjust the location of the output phase, it is possible to build a phase blender without one, so this is considered an "optional" input.

To better understand how DaVE handles optional inputs, Equation 2.3 shows the relationship for a basic amplifier with input `in`, output `out`, and no optional inputs. The DC model for this circuit has two parameters, A and B, which correspond to the slope (gain) and intercept (offset). When the tool has data for many input-output pairs, it is easy to use linear regression to find least-squares best-fit values for A and B.

$$\text{out} = A \cdot \text{in} + B \tag{2.3}$$

When optional inputs are added to the circuit, for example a current bias, `ibias`, each of these

parameters is replaced by a regressable function of the optional input:

$$A = c_1 + c_3 \cdot \text{ibias} \tag{2.4a}$$
$$B = c_2 + c_4 \cdot \text{ibias} \tag{2.4b}$$
$$\text{out} = A \cdot \text{in} + B \tag{2.4c}$$

In this case the we assumed that each parameter is linear with respect to `ibias`. The function does not have to be linear, however. Any function is acceptable as long as the coefficients can be found using linear regression, which is why we call the class of acceptable functions "regressable." In other words, the output must be linear with respect to the coefficients, even if it is not linear with respect to the optional inputs. Here is a nonlinear example:

$$A = c_1 + c_3 \cdot \text{ibias} + c_5 \cdot \text{ibias}^2 \tag{2.5a}$$
$$B = c_2 + c_4 \cdot \text{ibias} + c_6 \cdot \text{ibias}^2 \tag{2.5b}$$
$$\text{out} = A \cdot \text{in} + B \tag{2.5c}$$

We can substitute these expressions for A and B back into our original equation to find the circuit output as a function of the `input` and `ibias`:

$$\text{out} = (c_1 + c_3 \cdot \text{ibias} + c_5 \cdot \text{ibias}^2) \cdot \text{in} + (c_2 + c_4 \cdot \text{ibias} + c_6 \cdot \text{ibias}^2) \tag{2.6}$$

By multiplying through the parentheses in Equation 2.6, we can see that the entire right hand side is regressable and all six coefficients can be found with one application of linear regression. This is how DaVE is able to model circuits with arbitrary optional inputs affecting the behavior of the circuit.

DaVE splits optional inputs into three types: analog optional inputs, quantized analog optional inputs, and true digital optional inputs. In Equation 2.6 above, `ibias` would be considered an analog optional input because it has a value that can vary continuously. Next, we can add a quantized analog optional input, `adj[1:0]`. This optional input is physically quantized such that each bit of the input is a zero or one, but it still behaves like an analog input because the effect of each bit adds linearly with the effects of the other optional inputs. As a result, each bit of the bus gets its own term in the expression:

$$A = c_1 + c_3 \cdot \text{ibias} + c_5 \cdot \text{ibias}^2 + c_7 \cdot \text{adj}[1] + c_9 \cdot \text{adj}[0] \tag{2.7a}$$

$$B = c_2 + c_4 \cdot \text{ibias} + c_6 \cdot \text{ibias}^2 + c_8 \cdot \text{adj}[1] + c_{10} \cdot \text{adj}[0] \tag{2.7b}$$

$$\text{out} = A \cdot \text{in} + B \tag{2.7c}$$

Finally, we can add a true digital optional input. A true digital optional input completely changes the behavior of the circuit, so we use a separate copy of the equation with independent coefficients for each possible value of the true digital input. Here we add the `mode` optional input, and we see how it essentially doubles the existing expression with one copy corresponding to each possible value of `mode`:

$$A = c_1 + c_3 \cdot \text{ibias} + c_5 \cdot \text{ibias}^2 + c_7 \cdot \text{adj}[1] + c_9 \cdot \text{adj}[0] \tag{2.8a}$$

$$B = c_2 + c_4 \cdot \text{ibias} + c_6 \cdot \text{ibias}^2 + c_8 \cdot \text{adj}[1] + c_{10} \cdot \text{adj}[0] \tag{2.8b}$$

$$C = c_{11} + c_{13} \cdot \text{ibias} + c_{15} \cdot \text{ibias}^2 + c_{17} \cdot \text{adj}[1] + c_{19} \cdot \text{adj}[0] \tag{2.8c}$$

$$D = c_{12} + c_{14} \cdot \text{ibias} + c_{16} \cdot \text{ibias}^2 + c_{18} \cdot \text{adj}[1] + c_{20} \cdot \text{adj}[0] \tag{2.8d}$$

$$\text{out} = \begin{cases} A \cdot \text{in} + B, & \text{mode} = 0 \\ C \cdot \text{in} + D, & \text{mode} = 1 \end{cases} \tag{2.8e}$$

With these three types of optional inputs, DaVE is able to extend a template's description of a circuit to match the pinout of the user's circuit, and allow the user circuit's pins to affect the behavior of the modeled circuit in a realistic way. Fixture implements similar optional input types, which will be discussed in Section 4.3.

2.4.3 The DaVE Environment

The original DaVE tool is divided into three sub-tools: mProbo, mGenero, and mLingua. We will summarize these tools, and propose that our new tool, Fixture, should be considered a part of the DaVE environment. Figure 2.3 provides a visualization of the different tools within DaVE.

mProbo

mProbo is the portion of DaVE that deals with circuit characterization. mProbo begins with a user's description of the inputs and outputs of their model, and modifies the template equations according to the user's optional inputs as shown in the previous section. In order to collect the input-output pairs needed to fit a model, mProbo produces a testbench to be run on an external simulator, Cadence's Spectre AMS. The tool selects inputs to run such that each point is pseudo-randomly

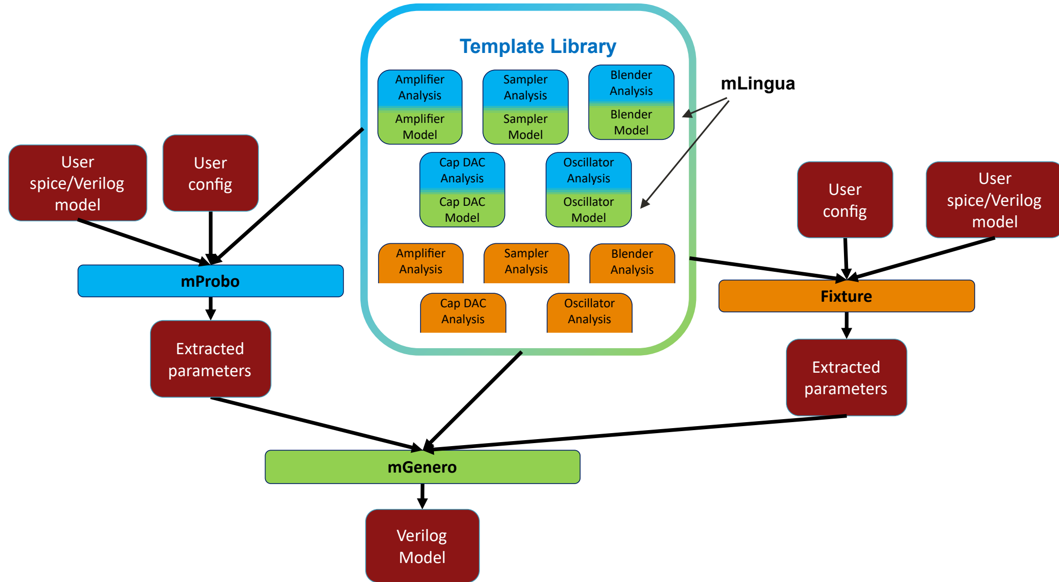


Figure 2.3: The different tools in the DaVE environment. The template library initially had pairs of templates, which were intended for use with mProbo (analysis templates) and mGenero (model templates). The mGenero model templates were written using the mLingua library. Fixture offers a replacement for mProbo, and correspondingly will need its own replacement templates in the library.

placed within the input space. With the smoothness assumption mentioned in the previous section, this allows for the entire input space to be characterized. The testbench itself is implemented in Verilog, and Spectre AMS is used to simulate either the spice or Verilog version of a circuit. Finally, mProbo extracts output points from the testbench results, converts them to the domain used for this circuit's central relationship, and uses linear regression to fit the parameters for this particular user circuit's central relationship.

mGenero

The use of a common form for the central relationship, either linear or piecewise-linear in each coefficient, is not only useful for extending parameter-fitting functionality but also for extending model generation functionality. mGenero is the portion of DaVE that can insert extracted parameters (usually found by mProbo) into templatized Verilog models. The templatized Verilog functional models are hand-written and mostly handle the domain conversion between the circuit's physical inputs and outputs and the domain where the tool can apply the central relationship. mGenero then fills in the details of the central relationship, including the effects of the user circuit's optional inputs.

mLingua

The final piece of DaVE is mLingua. This is a library of Verilog primitives that are extremely useful for real number modeling of analog circuits for CPU simulation. mLingua uses an event-driven piecewise-linear representation of time-varying signals. The timestep is dynamically adjusted to keep the error within a user-specified tolerance. mLingua is not just useful for DaVE tools; it can also be used to build models by hand or with other automated tools [19, 45] .

Other DaVE Capabilities

The main intended use of DaVE is automated model generation, but this functionality is built out of smaller steps that can also be useful in other ways. For example, mProbo can perform model comparison in addition to model generation. The exact same testbench that analyzed a spice circuit can be used to analyze an existing Verilog functional model instead, resulting in a set of parameters that describe the behavior of the Verilog functional model. Then, the sets of parameters can be compared between the spice model and the Verilog model to see whether they agree within an acceptable margin of error. This process can be performed on the Verilog produced by DaVE, or on a Verilog model produced externally. Through a chain of two steps, verifying that the model matches the spice circuit and verifying that the model works in a system-level testbench, the user can be confident that the spice circuit will perform correctly in a system-level context.

The model generation portion of DaVE can also be used on its own to generate a preliminary model of a circuit before the spice circuit has been written. Fixture extends this functionality by using a portion of the circuit analysis workflow to extract a model from a circuit specification (Section 6.5).

We hope that engineers continue to add new tools to the DaVE environment to add new capabilities to the library. In the next section, we will discuss how users can add circuit-specific functionality by contributing to the template library.

2.4.4 An Evolving Library

Why do we go through the effort of writing a Verilog functional model if we already have a spice-level model? As discussed in Section 2.1, the main reason is speed. Modern spice simulators are incredibly well optimized [46], so simply recreating the functionality of a spice model in Verilog will not gain us any performance. The key to writing a good model is to not simulate the behaviors that are not observed. To start, this means not calculating the voltages of internal circuit nodes *if* the model writer can accurately compute the output without them. If the circuit is an amplifier, there is no need to compute the currents and voltages of all its internal transistors; it only needs to accurately model the circuit's input / output relationship. The complexity of the model depends on how the circuit will be used. For example, if the amplifier is linear over the intended input range, then the

model can safely ignore the saturation behavior as long as it verifies (through Verilog assertions) that it is only presented with inputs in the intended range.

In summary, creating a good model can involve many simplifications without affecting the accuracy of the computed output, and a library with many options that can be enabled or disabled in each template will be more likely to create the circuit model the user is looking for. For this reason, DaVE allows and encourages users to contribute new templates or improvements to existing templates. Creating a template will always take more work than creating a single functional model, but DaVE hopes that users will recognize the long-term benefit of creating and sharing templates.

When considering how much time is saved by reusing an existing model it is important to remember the accumulated bug fixes and improvements that come with a well-tested piece of code. In the case of analog circuit modeling, the model-writer must contend with their own bugs in the analysis and modeling and also any mistakes that need to be uncovered in the original circuit and the surrounding testbench. A template can help with the former by being bug-free itself, and can also help with the latter by including the non-obvious circuit behaviors that analog or digital designers are likely to get wrong.

Consider the model of a phase blender. It seems like an easy candidate to model as a linear relationship plus domain converters. If the spice version of the phase blender is implemented and used correctly, this simple model will work perfectly well. However, if the digital control is mis-timed and causes glitches in the actual circuit, the simple model may not capture this behavior properly (Figure 2.4). This can hide bugs in the real circuit, leading to much bigger issues down the line. A properly-designed template would be aware of this glitching behavior and detect or include it in the model, even if the user had never considered it.

The ability to add circuit-specific tweaks to the existing library is helpful for circuit analysis as well as circuit modeling. Occasionally, there are circuit parameters that are easy to model but difficult to extract from testbench results. In these cases, there may be a more clever testbench setup that should be saved as part of the template. For example, a phase blender may or may not have significant output delay due to a buffer, but the periodic nature of the output makes it difficult to distinguish which input edge corresponds to which output edge. To determine the output delay, it is easiest to make a testbench where the input clock shuts off after some time, making it clear which edge corresponds to which (Figure 2.5).

In summary, the reusability in a template library is not only useful because it saves effort in writing the model, but also because it allows non-obvious improvements in a model to be saved and shared.

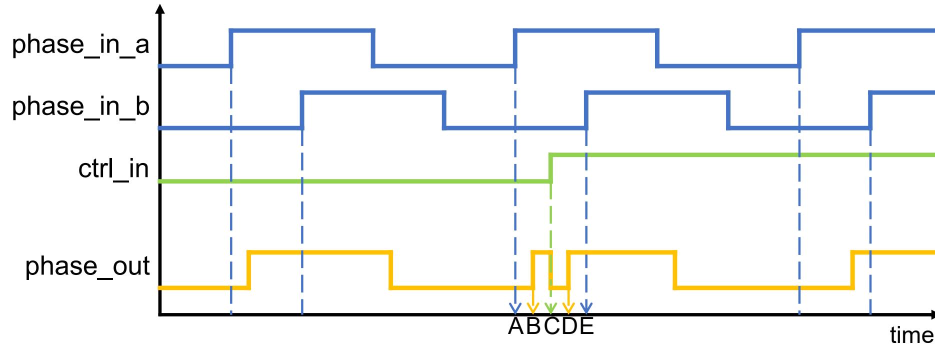


Figure 2.4: Diagram of phase blender glitching behavior. When the `ctrl_in` input is low, the `phase_out` edge is early, and when the control is high it is late. When the control switches sometime in the range of possible output times (referring to the labels on the time axis, when C occurs in the range between A and E) it is possible to observe early and late edges in the same cycle (edges occur at both B and D). In most cases the fix is not a change to the phase blender itself, but a change to the driving circuitry to prevent this case from occurring. It is the job of the analog functional model to mimic this glitch or raise an error flag when it occurs to alert the circuit designers of the issue in the driving circuitry.

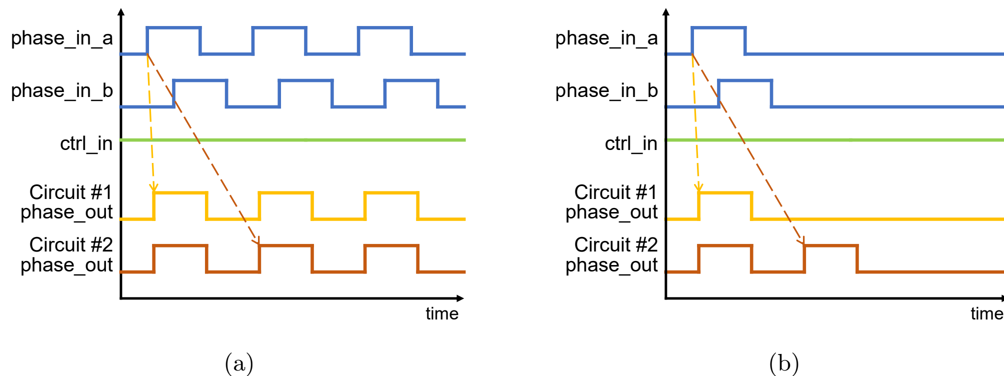


Figure 2.5: Each plot shows two different phase blender circuits responding to the same input stimuli. In Circuit #1, the input edges propagate to the output almost immediately. In Circuit #2 there is a delay equivalent to one period of the input clocks. Looking only at Figure (a), this difference is impossible to detect. In Figure (b), however, the input stimulus has been changed to turn off the input clocks after a certain number of cycles. With this stimulus is it clear that the Circuit #2 output is delayed by one period with respect to Circuit #1. A phase blender analysis testbench can make use of this strategy to easily determine which input edges correspond to which output edges for an arbitrary user's circuit.

Chapter 3

A Motivating Circuit

This thesis focuses on the development of the tool Fixture. Fixture is based heavily on DaVE tools, but extends the functionality of DaVE to handle more complex circuits. In this section we will assume some familiarity with DaVE tools as described in Section 2.4, including the "optional" vs. "required" input terminology. To discuss the improvements added by Fixture we will first describe a challenging circuit to model, then show where DaVE falls short in characterizing the challenging circuit, and finally turn these shortcomings into guiding principles for the design of Fixture.

For many of the challenges in this section the solution would be obvious if we were hand-writing the model for this specific circuit, but the difficulty is to generalize that solution so it extends to as many user circuits as possible and to have the tool apply the solution automatically to reduce the guidance needed from the user. Additionally, creating an accurate and efficient circuit model is not the only difficulty when designing an automated modeling tool. The user-friendliness of the tool's interface, the ease with which new users understand its operation, and the ability of users to debug model issues are all important features of the tool as well.

3.1 The Challenging Circuit

While working with the existing DaVE tools, we discovered several circuits that were difficult or impossible to model with the existing system. To motivate the need for Fixture we will combine the most challenging aspects from those circuits into a single amplifier and explore the issues associated with modeling that amplifier.

The example amplifier is a differential Transimpedance Amplifier (TIA) with several digital and analog trim inputs as shown in Figure 3.1. The implementation is a high-gain differential voltage amplifier with resistive and capacitive feedback. The resistive feedback is implemented with a bank of binary-weighted parallel resistors, controlled by digital input `radj[5:0]`. The capacitive feedback is implemented with a bank of binary-weighted parallel capacitors, controlled by digital

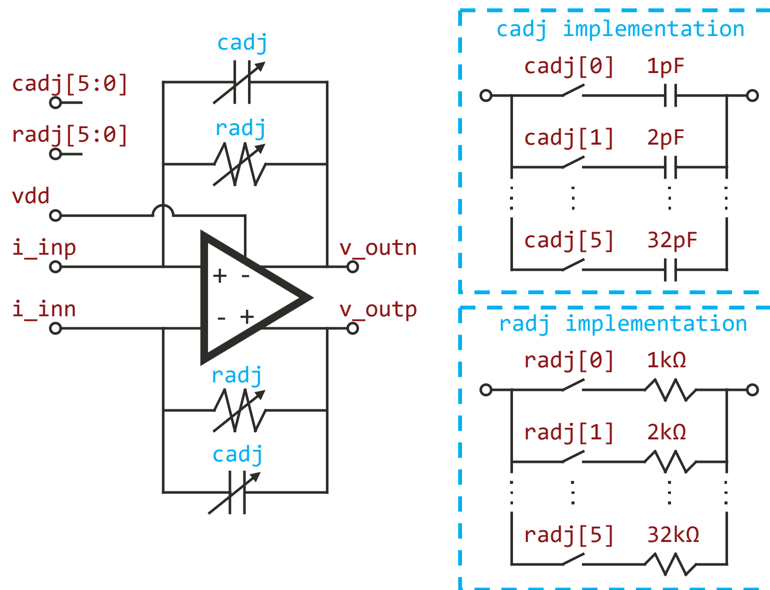


Figure 3.1: Challenging TIA circuit. In addition to the circulating current input and differential current output, the TIA has three other inputs that modify the behavior. `radj[5:0]` controls the adjustable feedback resistors, which set the gain. `cadj[5:0]` controls the adjustable feedback capacitors, which, along with the resistors, set the dynamic response. `vdd` is the supply voltage, which adjusts the common mode output level as well as other circuit parameters.

input `cadj[5:0]`. Notice that while the implementation of the capacitor and resistor look similar, the behavior is different because the components combine in parallel in different ways. For the capacitor the effective capacitance is proportional to the binary number defined by the bits. For the resistor, however, the effective resistance is *inversely* proportional to the value of the bits interpreted as a binary number in reverse order. This reciprocal relationship between binary value and resistance will lead to challenges with modeling this circuit. In DaVE and Fixture models are pin-accurate, so we include the supply voltage input `vdd` as well. In this particular circuit the behavior is also sensitive to the supply voltage, so that effect will be included in the model.

3.1.1 User input

The first step in automated model generation is collecting information from the user about the circuit to be modeled. Already, we encounter difficulties with the way information is communicated to the tool.

In order to report results in a domain familiar to the user, Fixture needs to know that the user wants to work in the differential / common-mode space for this differential circuit. The first challenge when modeling the TIA is its use of “circulating current” in the the user’s definition of gain. In a typical voltage-to-voltage differential amplifier, gain is defined as:

$$\text{gain}_{\text{from differential input}} = \frac{\text{v}_{\text{out,pos}} - \text{v}_{\text{out,neg}}}{\text{v}_{\text{in,pos}} - \text{v}_{\text{in,neg}}} \tag{3.1}$$

However, a different definition is sometimes used when the input current is viewed as a single current signal passing into the positive input and out of the negative input. In this case, the numerical value of the input is considered the value of this current, rather than the difference between positive and negative sides. With the circulating input current the gain equation becomes:

$$\text{gain}_{\text{from circulating current}} = \frac{(\text{v}_{\text{out,pos}} - \text{v}_{\text{out,neg}})/2}{\text{i}_{\text{in,pos}} - \text{i}_{\text{in,neg}}} \tag{3.2}$$

These competing definitions for differential gain present a problem. The tool will keep its definition consistent between analysis and modeling so the produced Verilog model will always behave correctly, but the reported value for differential gain would be off by a factor of two if the wrong definition is used. Since the modeling engineer likely knows what gain to expect this would lead to confusion and a search for the bug even though the produced model is functionally correct. The challenge in solving this problem is to produce a good system of user input so that the choice of definition is clear without being overly verbose. Fixture allows users to specify one signal as a linear combination of others with any coefficients, and the format for doing so is discussed in Section 5.2.2.

A second challenge with the tool's user input is the specification of loading effects. To properly characterize an analog block the tool needs to run its simulation with a realistic load on the circuit, which will depend on the context where the model will be used. First we will distinguish between "static" and "dynamic" loads: static loads do not change over the course of the circuit's operation, for example a Resistive/Inductive/Capacitive (RLC) network with fixed component values. Dynamic loads change over the course of a circuit's operation; this is most commonly the result of the following block changing its mode of operation.

There are many ways one could specify a static load to the simulator: capacitance, the `wire_load` model used by Liberty files [47], s-parameters, etc. The situation is even more difficult for dynamic loads. In DaVE, loads are handled differently by different templates. In Fixture, we would like to take a more unified approach. We allow the user to specify the load in spice so the user has total freedom over how the load is defined, and can even dynamically adjust the load based on additional inputs into the model (Section 4.3.5). Rather than create our own system for load specification, we allow users to specify the load in a language they are already familiar with.

A user circuit can have numerous optional inputs, and when each optional input affects each circuit parameter this leads to a very large number of coefficients describing the model. Most of the time, however, the analog designer intends for many of the coefficients to be zero because the optional inputs are designed to change one parameter with minimal side effects. To keep their model small and efficient the user may want to leave out these small coefficients, or to see whether small side-effects affect system behavior they may want to leave them in. Besides the challenge of producing

the best model, there is a tradeoff in the design of the tool between the simplicity of automatically modeling every effect and the flexibility of letting the user decide which effects to model. In DaVE all these effects were included in every model, but Fixture handles this issue by modeling the effects by default and allowing the user to specifically request otherwise (Section 4.4.2).

3.1.2 Choosing Sample Points

The TIA has many inputs to adjust its behavior and account for PVT variations. The generated testbench must exercise each of these inputs to ensure they have the correct effect in the model. Although the existing DaVE tools are designed to do this, we found some problems when modeling the adjustable resistance and capacitance of the TIA.

The TIA is designed with an adjustable resistance and an adjustable capacitance in the feedback path. The resistance directly sets the gain; however, some resistor values will correspond to gains above what the circuit designer intended. Ideally, the simulation should never drive inputs in a way not intended by the circuit designer. For the user to communicate the range of acceptable values, however, the tool needs to know both the decimal values of the intended limits and know how to convert the individual bits to that decimal representation. In our challenging circuit the bits should be interpreted as a binary number, but other encodings such as a thermometer code are also common. In DaVE, the tool had no awareness of the encoding of buses because every bus was modeled the same way, and this means there is no way to specify input ranges. Even when the full range of input values is acceptable, not knowing the encoding can still lead to issues; for example, choosing a random value for each input bit is very unlikely to choose extreme values for a thermometer-coded bus. In Fixture, we have the user specify the encoding in order to correctly distribute input samples, use the proper range of input values, and produce plots with user-interpretable axes (Section 4.3.3).

The product of the resistance and capacitance sets the time constant for the TIA's dynamic response. Generally, the intention is to set desired gain with the resistor value, then choose the corresponding capacitance to hold the time constant at its nominal value. The wide range of desired gains means that both the resistance and the capacitance need to be able to span a large range of values. Even though the analog designer only intends a small range of time constants to be used, the design means that a user of the circuit can select a wide range of time constants by choosing unintended input combinations. The range of intended capacitance values for a given resistance value is somewhat difficult to specify because it is a nonlinear function of the resistance value. Unfortunately, simulating and modeling unintended combinations of resistance and capacitance can lead to several issues: if the capacitance is too small the closed-loop time constant could be set by the open-loop amp, which should never happen during normal operation; if the capacitance is too large the simulation could take too long to settle, either reading unsettled output values or wasting computation time on a long-running simulation. For these reasons, the existing DaVE approach of sweeping the entire input space is inadequate for testing the TIA. Fixture addresses this challenge by

allowing the user to specify an acceptable relationship between the chosen resistance and capacitance values (Section 5.1.3).

3.1.3 Simulation

One of the key attractions of DaVE and Fixture is that they are open-source. They do, however, interact with simulators that may not be open-source or freely available. In order to compare spice circuits to their Verilog models the tool needs to run the same testbench on both versions of the circuit; DaVE tools solved this issue by using an AMS simulator, Spectre AMS. Compared to spice simulators and Verilog simulators individually, however, AMS simulators are more complex and there are fewer simulators available. The limitations of always using an AMS simulator were especially apparent when we worked with the open-source Sky-130 PDK, as Spectre-compatible models were not available at the time. As a result, we were not able to use DaVE circuit characterization at all for that project. When making models by hand it is possible, though time-consuming, to write separate spice and Verilog testbenches and choose compatible simulators for each when no AMS simulator is available. Fixture is able to achieve that same flexibility by using the *fault* library to compile testbenches to either spice or Verilog format (Section 5.2).

3.1.4 Model fitting

After collecting data for circuit behavior at a variety of optional input operating points DaVE and Fixture attempt to find expressions to model each of the parameters in terms of the optional inputs. The DaVE tools model each parameter as a linear or piecewise linear function of the input pin values. If a particular parameter is not being modeled to the required level of precision, the usual approach is to switch to a piecewise linear model and increase the number of pieces until the desired precision is met. DaVE can also add terms which are polynomial functions of one or more inputs, which can behave like a Taylor Series to theoretically match any desired curve. Although these approaches have a lot of expressive power, we still had issues when trying to model the TIA. We will describe the issues with modeling the gain and time constant, as well as the effect these issues had on the overall debugging experience.

The gain of the TIA is set directly by the feedback resistor. This resistor is implemented as a bank of binary-weighted resistors arranged in parallel, each of which is connected or disconnected based on a digital control bit (Figure 3.1). It is straightforward to come up with the following expression for the gain:

$$\text{gain} = \frac{1}{\sum \text{radj}[i] \cdot \frac{1}{\text{R}_i}} \tag{3.3}$$

Where $\text{radj}[i]$ is the state of the ith digital control input, and R_i is the value of the ith resistor in the bank. When the resistor values are known, this expression is easy to understand and easy

to implement in nonsynthesizeable Verilog. While a piecewise linear function of the control inputs might reach the desired level of accuracy with four or eight pieces, such a model would not be as understandable or as accurate as the simple nonlinear expression in Equation 3.3. For this reason, the nonlinear model is preferable to model designers. One of the main improvements Fixture makes over DaVE is the ability to model effects using arbitrary nonlinear expressions.

Further issues with the existing modeling options are clear when trying to model the RC time constant of the TIA. We will ignore for a moment the difficulty of modeling the resistor in the previous paragraph, and assume that a linear model is still within the user's desired error tolerance. In this case, the 6-bit resistance can be modeled with 7 parameters (one coefficient for each input bit plus an offset), and the capacitance can be modeled with another 7 parameters. The time constant for the TIA is simply the product of these two values, and can therefore be modeled with 14 parameters. In DaVE, however, this product-of-sums is not an allowable model because the 14 parameters cannot be fit using linear regression. The only way to model the product of these two values is to consider a coefficient for each cross-term between the two input buses. We illustrate this concept with 2-bit versions of the resistor and capacitor inputs, using v_i for the resistor input vector, w_i for the capacitor input vector, and c_i for the coefficients. In Equation 3.4a we show the desired relationship, and in Equation 3.4b we show the equivalent representation in DaVE:

$$RC = (c_1 \cdot \mathrm{v}_0 + c_2 \cdot \mathrm{v}_1 + c_3)(c_4 \cdot \mathrm{w}_0 + c_5 \cdot \mathrm{w}_1 + c_6) \tag{3.4a}$$

$$RC = c_1 \cdot \mathrm{v}_0 \cdot \mathrm{w}_0 + c_2 \cdot \mathrm{v}_0 \cdot \mathrm{w}_1 + c_3 \cdot \mathrm{v}_0 \cdot + c_4 \cdot \mathrm{v}_1 \cdot \mathrm{w}_0 + ... \tag{3.4b}$$

Unfortunately, the cross-term version of the expression (Equation 3.4b) has on the order of N^2 terms for N-bit buses. For our 6-bit resistance and capacitance banks the expression balloons to 49 terms, and the resulting 49-parameter model is much more general and therefore requires more data to find a reasonable fit. This is not a viable way to model the time constant. In Fixture, we can use the arbitrary expression modeling (Section 4.4.2) to use the product-of-sums form of the expression (Equation 3.4a).

Finally, we come to an issue with debugging the tool's model fitting. Over the course of trying to model the TIA, the linear regression would often fail to find coefficients that resulted in an accurate model. The root cause of the issues were the shortcomings of the linear and piecewise linear models described above. But at first this root cause was not obvious, and the linear regression results did not offer any information to help find the issue. Ideally, in the situation where the tool is able to accurately model the `vdd` effects but not the `radj` effects, one would hope that the resulting model would contain the correct coefficients for `vdd` and the best possible coefficients for `radj`. Unfortunately, the tool does not have the information it needs in order to do that. Recall that to get good coverage of the circuit's behaviour with minimal simulation, the tool sweeps all the inputs simultaneously. The tool has no way to separate changes in the output due to `vdd` from changes due

to `radj`. Therefore, it cannot find the ideal coefficients for `vdd` because they are obscured by the "noise" of `radj` effects that cannot be modeled with the given form of the modeling expression. The effect on the user is that they receive a model that doesn't meet the error tolerance, and receive no guidance on which input has a nonlinear effect that is causing the model to fail. Fixture addresses this issue by picking inputs points in such a way that the effects of different optional inputs can be separated (Section 6.1.4). Additionally, Fixture produces plots that allow the user to see the effects of individual optional inputs and judge their models separately (Section 6.2.2).

3.2 Guiding Principles

Throughout this thesis, we will address each of the issues outlined in this chapter and build Fixture so that it can correctly handle these challenging cases. Rather than implement one-time fixes for each issue, we try to categorize the challenges and come up with general principles that would have prevented them in the first place. With this approach, we hope to catch future issues before they arise.

3.2.1 Provide Reasonable Defaults, but Advanced Options

The first principle is to have reasonable defaults, but advanced options for the various features of the tool. After modeling several challenging circuits it is clear that there are many reasons the tool may not be able to produce a quality model without additional information from the user. Unknown test environments, non-standard circuit configurations, and speed-vs-accuracy tradeoffs all necessitate help from the engineer. However, asking for the engineer about all these factors explicitly is unnecessary in the majority of cases, and this extra work at the start can make it difficult for new users to adopt the tool. We aim for the middle ground, where the tool has default settings that work most of the time, but advanced abilities that can be utilized when needed.

There are a few difficulties with providing default options. The first is that a middle ground can be difficult to achieve because different users may have different opinions on what the default should be. Second, users may not know that a solution to their problem already exists in the tool if that solution is hidden away as an "advanced" option. Fixture can mitigate these first two issues to some extent by providing a library of examples. This helps us estimate which options are the most common and also provides new users with examples of each feature being used. The biggest issue, however, is the danger of defaulting to an option that produces incorrect output. For example, if the tool silently defaults to a differential input when the user assumed a circulating current input, then it can be difficult for the user to figure out why the reported gain is half what they expected. For each option we must be sure to either make the user's choice clear, or produce output that clearly demonstrates what the tool defaulted to.

This guiding principle works well with the idea of an evolving template library. As templates

gain new features the writers can simply add them to a growing list of advanced features that users have access to. In some cases, such as a test that can catch easy-to-miss bugs in an analog design, the writer of the new feature may also want enable it as a default. This can be more challenging, as they must be aware of the effect this has on existing uses of the template in terms of run time, correctness of output for a variety of circuits, and clarity to the user on what the new default is doing.

3.2.2 Follow a Process that is Familiar to Engineers

The next guiding principle is to follow a process that is familiar to human engineers. Many engineers already build functional models by hand, and have established a process for doing so. We found that it is generally best to follow this existing process as closely as possible. In some cases this makes the tool easier to write; for example, we let the user attach a load to their circuit inside of spice rather than create our own format for specifying the load. In other cases this makes the tool more difficult to write; for example, we compile testbenches to both spice and Verilog rather than using an AMS simulator to reuse the same testbench for all models.

One reason to follow familiar approaches to modeling is that new users can learn the tool with less effort. This makes it easier to get more people using and contributing to the tool, and also reduces the number of mistakes made due to misunderstanding of the tool's process. Additionally, the modeling processes used by engineers are already the result of trial-and-error and debugging, so following these processes can avoid unforeseen issues. In other words, trying to deviate from the usual process can cause more issues than it solves. For example, while the tool can use linear regression to fit the effects of many optional inputs at once, a human model designer usually simulates the effects of one optional input at a time. Originally the tool was able to reduce the amount of simulation by never sweeping optional inputs individually, but we found that changing the tool to perform individual sweeps in addition to simultaneous sweeps provides several benefits at once. First, it provides much-needed debugging information to the user when the specified modeling expressions cannot model the circuit behavior. Second, that same change helps produce more accurate parameter-vs-input plots (Section 6.2.2). Finally, it also aids the nonlinear optimizer in finding the right coefficient fits (Section 6.1.4). By following this guiding principle we make sure to leverage the progress that has already been made in the field of analog modeling.

The biggest roadblock to following this guiding principle is that it is not always obvious how to generalize existing approaches. For example, an engineer modeling our motivating circuit may decide that the best way to model the gain is to assume it is proportional to the inverse of the `radj` bus value. But how should the tool automatically determine the order of bits in the bus? How does that strategy generalize to a different circuit that uses a bias current to adjust gain values? What if the resistor values are not perfectly binary weighted? Answering these types of questions is possible, but it takes a lot of trial-and-error with many real-world circuit examples.

3.2.3 Make Data User-Accessible

There are some circumstances where the tool is dealing with a large amount of data or data in a format that is difficult to interpret. When everything goes smoothly the user may simply accept the final output and never consider these esoteric internal steps. But if the user wants greater trust in the output or needs to debug an issue with the final model then they may want to follow and understand each step individually. For this reason, our final guiding principle is to make any data in the tool as easy as possible to observe and interpret. At a minimum, this means exporting the results of each step (chosen input points, compiled testbench, raw testbench results, processed testbench results, fitted model coefficients) to an easily-processed format so the user can do their own analysis. When possible, the tool should also present the data in a more human-interpretable format such as a plot. For example, in the case where the TIA is only intended to use certain combinations of resistance and capacitance settings, a plot of the input combinations being simulated would quickly show a user whether the inputs are being sampled correctly or not. When fitting model parameters, a plot of collected data vs. modeled outputs can quickly give the user an idea of whether the model is within error tolerance, and suggest whether the error is random or systematic. In general, improvements to user-readability may be extra effort to implement in the tool, but save time on the user's end because they can more easily follow what the tool is doing.

Chapter 4

A Template Library

In Section 2.4 we discussed what a template library is, and why we decided to build our own. In this section we will dive in to the specifics of how templates are implemented in Fixture. Designing a good template structure requires a balance between giving the template writer freedom and constraining the template writer to a structure so that the tool can extend the template automatically. Giving the template writer more structure can also be helpful if it can guide a new writer through template creation. In Section 4.1 we will describe the particular structure and constraints we chose for templates in Fixture. In the remaining sections we will describe the many ways Fixture can automatically extend templates, all of which rely on the specified structure.

4.1 Structure of a Fixture Template

Writing a template for a new type of circuit is not a simple task. Besides determining which analog effects need to be modeled, the template writer needs to provide testbench, analysis, and modeling instructions for each effect. Fixture helps the template writer with this large task by breaking it into smaller pieces and by handling as much of the work as possible automatically. This section will walk though each of the pieces the template writer needs to implement and describe its purpose. We will use a simplified amplifier template as a recurring example throughout this section; please refer to Figure 4.1 for a visual representation of this template.

The first way of breaking the template into smaller pieces is organizing the important circuit behaviors into tests. A template can have any number of tests. Each test has a different set of circuit parameters it can extract, and each test can run independently of the others. The core feature of a test is its simulation testbench, so the main reason to separate two circuit behaviors into different tests is because the template writer wants to use different input stimuli to analyze those behaviors. For the simplified amplifier template we break the simulation into a DC Test, for measuring the DC transfer function, and a Dynamic Test, for measuring the poles and zeros of the amplifier. The

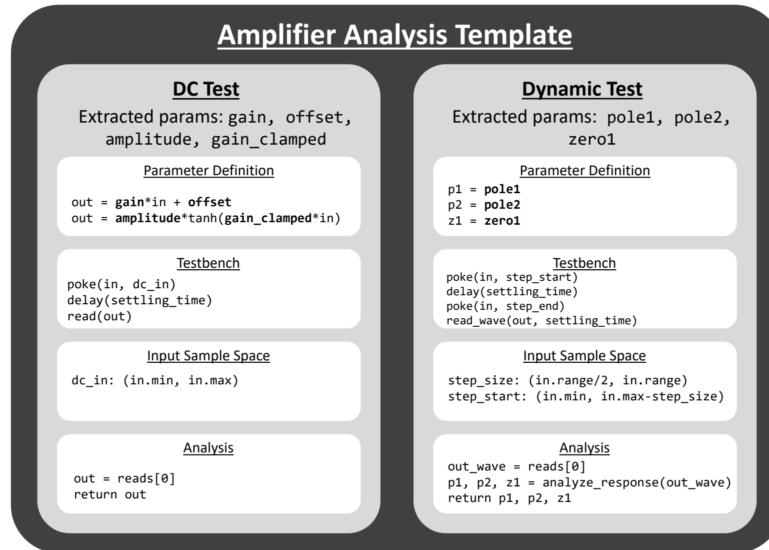


Figure 4.1: Simplified version of the amplifier template, showing organization into two tests (DC Test, Dynamic Test) and the four sections within each test (Parameter Definition, Testbench, Input Sample Space, Analysis).

output of a test is a set of coefficients; the exact set of coefficients and their meaning is a combination of the way the test writer decides to describe the behavior and the optional inputs specified by the user.

The user can decide which tests they want to run depending on which circuit behaviors they need to model and how much simulation time they are able to dedicate. The reason we allow users to select which analyses to do at the test level, rather than the parameter level, is because simulation is the most computationally expensive part of the process. We are essentially allowing the user to decide which simulations to run, and tests are always designed to extract as many parameters as possible from that simulation. Unless the user specifically disables some tests, Fixture runs all tests by default to give the user the most information possible about their circuit and let them see which parameters are useful for their particular circuit. This follows our guiding principle of defaulting to a reasonable option, but allowing the user to tweak the functionality with advanced options.

Once the template writer has decided how to break the analysis into individual tests, they must implement each test. Since the high-level procedure for extracting parameters is the same each time, Fixture handles much of the work automatically and only exposes the test-dependent parts to the template writer. The parts the template writer needs to implement are broken into four separate sections. We will walk through the four sections using the simplified amplifier template as an example (refer to Figure 4.1).

Parameter Definition

The first step in creating a test is to define the parameters that will be extracted. A parameter is defined by the way it relates inputs and outputs of the circuit. The template writer simply needs to write these relationships as equations, following some rules discussed in Section 4.2. Looking at the Parameter Definition section of the DC Test in Figure 4.1 we can see two equations, which define a total of four parameters. The first equation corresponds to a linear model of the amplifier, and that linear model is uses the parameters `gain` and `offset`. The second equation corresponds to a nonlinear version of the amplifier which clamps the output with a `tanh` function. It defines the parameters `amplitude` and `gain_clamped`.[1] Note that although the test extracts all four parameters every time, a Verilog model would not use all four of these parameters; it would choose which ones to use depending on how it wants to model the amplifier circuit.

Besides the parameter names, the DC Test equations also use the names `in` and `out`, which refer to the values of the corresponding pins of the amplifier template. The Dynamic Test, however, does not use these values directly and instead uses values derived from those pins. Specifically, `pole1`, `pole2`, and `zero1` are calculated based on the `out` waveform. The Analysis section will describe the process for defining these derived values. We refer to the direct template pin values and derived values together as "measured values" to differentiate them from the parameters in the parameter definition equations.

Testbench

The Testbench piece of a test describes the input stimuli to apply to the circuit and the outputs to read in order to collect one measured datapoint. Fixture will repeat this short testbench many times, possibly while varying other circuit-specific inputs, in order to collect many measured datapoints. In the amplifier DC Test the Testbench simply applies an input voltage, waits for the amplifier to settle, and reads the output. In the Dynamic Test, the Testbench sets the input to a beginning step value, then applies the step and reads the entire waveform of the output as it is settling. This section only defines the testbench, and it is not compiled and run until later, so any post-processing of the results needs to wait until the Analysis section. The Testbench piece of the test is described in much more detail in Section 5.3.

Input Sample Space

When writing the Testbench, the template writer may find that they need a random number or a number that varies between different datapoints being collected. In the amplifier template, the DC Test needs to choose the input DC level to apply to the input. The Dynamic Test needs to choose both the step size and the start value for the input step it applies. Rather than use a pseudorandom

[1] To model a single-ended amplifier the nonlinear equation would also need an `offset` parameter, but it is omitted here for brevity.

function in the Testbench, Fixture has the template writer declare these variables in the Input Sample Space section. Later, when running the Testbench section, Fixture will pass sampled values for these variables into the Testbench. This allows Fixture to guarantee certain properties about the randomness that can improve the coverage of possible circuit behaviors (discussed in Section 5.1). When the template writer is declaring these variables they have access to properties of the user amplifier, such as the maximum and minimum values to apply to the amplifier input, and they use these to tailor the limits of the variables.

Analysis

The final piece of the test is the Analysis section. In this section the template writer receives raw values[2] from the simulation and transforms them into measured data to be used in the parameter definition equations. In many cases no processing needs to be done at all. This is the case for the amplifier DC Test because the parameter equations simply use the raw value of the amplifier's input and output pins. In the case of the Dynamic Test, however, the parameter equations use the values of pole and zero locations, which are not directly measured by the transient simulation. In the Testbench section the template writer measured the waveform of the output step response, so that is the information received in the Analysis section, and the template writer needs to extract the pole and zero locations. The Analysis section is written in Python, so the template writer is encouraged to use existing libraries and share common functions between templates.

4.2 Parameter Equations

In Section 4.1 we introduced parameter equations in the Parameter Definition section of a test. In this section we will describe the rules surrounding parameter definition, which ensure that Fixture is able to extend these equations to take optional inputs into account. Although these rules add some structure to the way the template writer can model a circuit, we try to allow as much freedom as possible in the definition of the parameter equations. We find that the rules surrounding parameter equations rarely limit the template writer, and more often it is the rules surrounding Testbench writing (Section 5.3) that are more challenging to deal with.

There is significant variation in the way parameters are used to describe circuits. The simplest is a function containing parameters that directly calculates circuit output as a function of circuit input.

$$\text{out} = \text{gain} \cdot \text{in} + \text{offset} \tag{4.1}$$

Equation 4.1 shows a very simple relationship with two parameters, `gain` and `offset`. This

[2] By the time the simulation values arrive in the Analysis section, they may have already gone through one stage of processing to transform from the physical domain to a custom domain defined by the user, as discussed in Section 4.6.

relationship might be used for a linear, single-ended amplifier. Every parameter equation shares this format with a single output on the left and a function of parameters and measured data on the right. The left hand side and the measured data can be raw circuit inputs and outputs, and can also be pre-processed functions of inputs and outputs (pre-processing happens in the Analysis section of the template). This format was chosen so that the same equation can be used directly in a Verilog model to calculate outputs, and also so that Fixture can produce plots with the left hand side as the dependent axis. It is worth noting that the Verilog model is not required to use these parameter equations directly. Fixture will simply extract values for the parameters, and models can use those values however they need to.

It is important to note that parameter equations never have time dependence themselves (such as a Laplace-domain transfer function might). If the template writer wants to describe a time dependence they must describe it using parameters representing delays, pole/zero locations, or other timing information. The amplifier Dynamic Test uses poles and zeros to describe the dynamic response of the amplifier, and defines three parameters with the following equations:

$$\text{p1_meas} = \text{pole1} \tag{4.2a}$$

$$\text{p2_meas} = \text{pole2} \tag{4.2b}$$

$$\text{z1_meas} = \text{zero1} \tag{4.2c}$$

Equations 4.2a-c each map a measured pole/zero frequency (on the left hand side) to a parameter representing that frequency (on the right hand side). The parameters in these equations are intended to be used in addition to something like Equation 4.1 to describe both the DC and dynamic behavior of the amplifier. Although these equations are just identity functions, recall that the left hand side represents many measured datapoints while the right hand side is a single parameter value fit to those datapoints. Additionally, this equation is only a starting point and can be transformed and expanded to match the needs of the user circuit (Sections 4.4 and 4.5).

So far, all the equations we have discussed in this section have been linear in the data and parameters. There is no requirement that the equation is linear; only that the equation is smooth (continuous) with respect to the data and parameters. The tool does not enforce or detect whether the equations are smooth, but encourages template designers to make them that way because Fixture assumes that circuit behavior varies smoothly between measured datapoints. As discussed in Section 2.4.2, if the template designer is observing a discontinuous curve they are likely considering the problem in the wrong domain. In Equation 4.1 we gave a parameter equation for a linear amplifier; now we will consider two nonlinear alternatives. The resulting transfer functions are plotted in Figure 4.2.

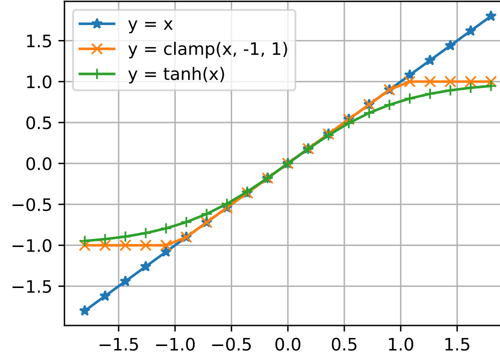


Figure 4.2: Comparison between three possible nonlinear differential amplifier models, which can be achieved with Equations 4.1, 4.3, and 4.4 respectively. The user can choose between these models depending on their speed and accuracy needs.

$$\text{out} = \begin{cases} \text{v_min}, & \text{gain} \cdot \text{in} + \text{offset} < \text{v_min} \\ \text{v_max}, & \text{gain} \cdot \text{in} + \text{offset} > \text{v_max} \\ \text{gain} \cdot \text{in} + \text{offset}, & \text{otherwise} \end{cases} \tag{4.3}$$

In Equation 4.3 we clamp the linear output voltage from Equation 4.1 between maximum and minimum values `v_max` and `v_min`. This is useful if the amplifier saturates at maximum and minimum voltages. Notice that the output remains continuous at the breaks between pieces for all possible values of the parameters. Notice also that several of the parameters appear multiple times in the right hand side of the equation; they are still assigned a single value which is used each time they appear in the equation. In some amplifiers the saturation is more gradual, and the user may not be satisfied with the sharp corners of the piecewise model. In that case, the template could fit a `tanh` curve to the amplifier's DC response.

$$\text{out} = \frac{\text{v_max} - \text{v_min}}{2} \cdot \tanh\left(\frac{2 \cdot \text{gain}}{\text{v_max} - \text{v_min}} \cdot (\text{in} - \text{in_offset})\right) + \frac{\text{v_max} + \text{v_min}}{2} \tag{4.4}$$

Equation 4.4 uses a tanh function to model the gradual saturation of an amplifier's output. Although this model may be more accurate, it is also more computationally expensive than the linear or piecewise linear models. Users may prefer to use one model over the others depending on the situation, so if the template contains all three, Fixture will fit all the options and allow the user or model writer to choose which parameters to utilize based on the fitting error of each equation.

4.3 Optional Input Types

Optional inputs, as described in Section 2.4.2, are the main way that Fixture is able to extend templates to fit different user circuits. In this section we will discuss the different types of optional inputs and explain exactly which circuits can and cannot be described using them.

4.3.1 Pinned Values

The models produced by Fixture are mostly pin-accurate to the original circuit, with exceptions for dynamic loading (Section 4.3.5) and non-electrical inputs (Section 4.3.6). This means that inputs are still included in the model even when they do not affect the behavior of the circuit, and even when they are expected not to change over the course of the simulation. In Fixture we refer to these fixed-value inputs as Pinned Optional Inputs. A common example is a supply voltage input `vdd`, when the user is not interested in modeling supply voltage variation. In this case the Verilog model will still have a `vdd` input, and there will be an assertion checking that the testbench applies the correct voltage within a certain percentage variation, but it will have no other effect on the model's behavior. We believe that including these inputs can prevent miscommunication between the model designer and the model consumer [10]. For example, if an analog designer changes the intended value for `vdd`, they must update the Fixture configuration to get the expected simulation results, and Fixture will automatically update the assertion on the value of `vdd`, which forces the testbench writer to make that update as well.

4.3.2 Analog

Most voltages or currents that affect the behavior of a circuit are considered Analog Optional Inputs. Common examples are adjustable supply voltages and adjustable bias currents. To be a valid Analog Optional Input, the input must meet two conditions: the circuit's output must change as a smooth function of the input, and the effects of multiple optional outputs must *mostly* combine linearly. The first condition, that circuit output changes as a smooth function of the optional input, is the hallmark of an analog circuit, and matches the smoothness assumption from Section 2.4.2. To understand the second condition, and why we only require linear addition only for "most" optional inputs, we need to consider how parameters are modeled with respect to optional inputs. We looked at one example already during the discussion of DaVE (Equation 2.8), but this time we will take a more general approach. Consider a single parameter being affected by multiple optional inputs, for example a gain that is affected by both `vdd` and `ibias` inputs.

$$\mathrm{g} = \mathrm{g}_{\mathrm{nominal}} + f(\mathrm{vdd}) + g(\mathrm{ibias}) \tag{4.5}$$

Although we allow f and g to be nonlinear functions of the optional inputs, they must affect the

gain in an additive manner. This allows us to consider the effects of `vdd` and `ibias` independently; changing `vdd` from a value of a to a value of b will always increase the gain by $f(b) - f(a)$, regardless of the value of `ibias`. This separability is helpful for data collection and parameter fitting (discussed further in Section 6.1.4). Additionally, this follows our guiding principle to follow the process already used by engineers, since human engineers will start by considering the effect of each optional input individually first. In cases where the effects of two inputs do not add linearly, we do allow an additional "cross-term" effect[3]:

$$\mathrm{g} = \mathrm{g}_{\mathrm{nominal}} + f(\mathrm{vdd}) + g(\mathrm{ibias}) + h(\mathrm{vdd}, \mathrm{ibias}) \tag{4.6}$$

Typically this combined function $h(\mathrm{vdd}, \mathrm{ibias})$ is more difficult to characterize: if f and g took on the order of N input-output measurements to characterize, h will take on the order of N^2. This is why we require most optional inputs to have linearly independent effects; a few combined functions will be okay to model but interaction between three or more inputs may quickly require more data than we can feasibly simulate. In this sense Fixture does not require that optional inputs have effects that add linearly, but it cannot guarantee to handle circuits if they do not follow that convention. Fitting with our guiding principle of reasonable defaults but advanced options, Fixture will not include any cross terms by default but does not stop users from explicitly including them in a model. More information about how the user specifies these dependencies is given in Section 4.4.

4.3.3 Quantized Analog

Quantized Analog Inputs were also used in DaVE, but one update we make from DaVE's approach is that we require the user to specify the type of bus. In Fixture we rely on the bus type to compute input samples (Section 5.1.2) and compute axes for debugging (Section 6.2.2). Additionally, only some types of buses have the property that each bit effects the circuit behavior independently (we will see that Signed Magnitude buses do not have this property), so the bus type is needed to properly choose a parameter's default dependence on the bus. Although specifying the bus type is a little bit of extra work for the user, we believe it is worth it in order to follow the process used by engineers already, which is one of our guiding principles. This is a case where we decided not to provide a smart default because both binary and thermometer codes are both commonly used, and more importantly because the bitwise-coefficient model works for many bus types, preventing Fixture from automatically detecting a mis-labling with a poor fit.

Quantized Analog Optional Inputs have similar behavior to Analog Optional Inputs, but the physical circuit inputs are binary-valued. Examples are any inputs that use a binary-encoded or thermometer-encoded bus, which is becoming more common as more analog circuits rely on digital trim bits [2]. The requirement that the output changes as a smooth function of the input is relaxed for

[3]Because h is an arbitrary equation we could fold the effects of f and g into it, but we leave them separate to match the intent of circuit designers who treat the cross term as an error on top of intended individual behaviors.

quantized analog inputs because the inputs themselves do not vary smoothly; each one is quantized to two possible values. The second requirement, that effects of multiple inputs add linearly, does apply to quantized analog inputs. Depending on the type of bus, the linear independence might apply on a bit-by-bit basis, or it might only apply to the effect of a bus as a whole. We will discuss the different bus types individually.

Binary, Thermometer, and Segmented

Binary coded buses are probably the most common type of quantized analog input. Each input bit acts like one bit of a binary number, so the total effect of the input is proportional to the binary number encoded by the bits of the bus. Thermometer codes are similar, except the weight for each bit is the same, so the effect of the bus is proportional to how many bits are one. Segmented encodings are a mix between binary and thermometer, with some bits being binary-weighted and others (usually the high-order bits) having equal weight. The weights for the bits in a segmented bus might be [8, 8, 8, 4, 2, 1].

These three bus types are similar because in each one the effects of individual bits add linearly with each other. Because of this, each bit could be sampled and modeled as an independent Optional Input. However, in cases where these models do not apply perfectly it can be easier to see the imperfections if the sampling strategy is tailored to the type of bus. Specifically, we want to be sure to simulate extreme circuit behaviors as much as mid-range behaviors, and this does not happen when choosing the bits of a thermometer-coded bus independently (Section 5.1.2). Additionally, we use the bus type information to present plots where axes correspond to the decimal values of the input buses, just as the user would expect (Section 6.2.2).

Signed Magnitude

Buses using a signed magnitude encoding scheme are also considered quantized analog. Although they are not as common as binary or thermometer, they are sometimes used, for example, to set a current and then steer it to the positive or negative side depending on the sign bit. Unlike binary encoding the effect of each bit in a signed magnitude encoded bus does not add linearly with the others. In other words, the value assigned to a particular bit pattern is not just a weighted sum of the bits that are high. One way to fix this for a signed magnitude bus is to use a weighted sum of bits and cross terms between the sign bit and each of the other bits - this is functionally equivalent to having separate weighted-sum models for the positive and negative cases. With Fixture we decided not to take this approach. Instead, Fixture allows parameters to be a function of the value of the bus as a whole, and uses a nonlinear function of the bits:

$$\text{bus_value} = (2 \cdot \text{bits}[i_{\text{sign}}] - 1) \cdot \sum w_i \cdot \text{bits}[i] \tag{4.7}$$

Even though it is not linear with respect to each bit, we still consider this a valid optional input as long as the effect of the bus as a whole adds linearly with other optional inputs.

Custom Functions

While working on Fixture we did not come across any digital encoding schemes for circuit inputs besides the ones we have already covered in this section. The strategy used for the signed magnitude encoding, however, is easily adaptable to any encoding scheme, so Fixture allows the user to define a custom encoding if necessary. The process for this is the same as the process for defining arbitrary Optional Input expressions, and will be covered in Section 4.4.2.

4.3.4 True digital

True Digital Optional Inputs are handled the same way in Fixture as they are in DaVE: the circuit is re-characterized for each possible digital mode. The only difference is that Fixture allows the user to specify a subset of 2^N possible modes for N True Digital inputs, and specify those as the only allowable modes. Fixture will only characterize this subset, and will add assertions to the Verilog models to ensure that only this subset is used in the system. This change can save significant simulation time in cases where there are a large number of True Digital inputs, but the user already knows exactly how they will be set when the system uses the circuit as intended.

4.3.5 Load Specification

In Section 3.1.1 we briefly discussed the challenge of load specification. Recall that in Fixture, circuits should always be analyzed under the same load they will have in practice. The library used by Fixture to manage simulations, *fault* (Section 5.2), has the ability to specify a capacitive load value to be added to each output for simulation. For many circuits, this will not be sufficient because the circuit needs a different load on each output, or needs a resistive load, or needs the load to change during operation. For this reason, Fixture does not rely on *fault* to set loading effects and instead has the user specify loads in spice directly.

In some cases the user can avoid modeling complicated loads by breaking the analog circuit into larger pieces. For example, an amplifier followed by a filter could be represented by a single dynamic amplifier model, and then the loading effects on the amplifier stage are inherently captured in that model and do not need to be explicitly characterized. This strategy only works if Fixture has a template that can represent the combined blocks.

If the user does want to model a block with a complex load, they must instantiate it in the spice file that will be passed to Fixture. The best way to do this is to create a wrapper for the circuit that includes the load, and have Fixture characterize the wrapper. In some sense this is exactly what we want to model in Verilog: because each model produced by Fixture is feed-forward, its

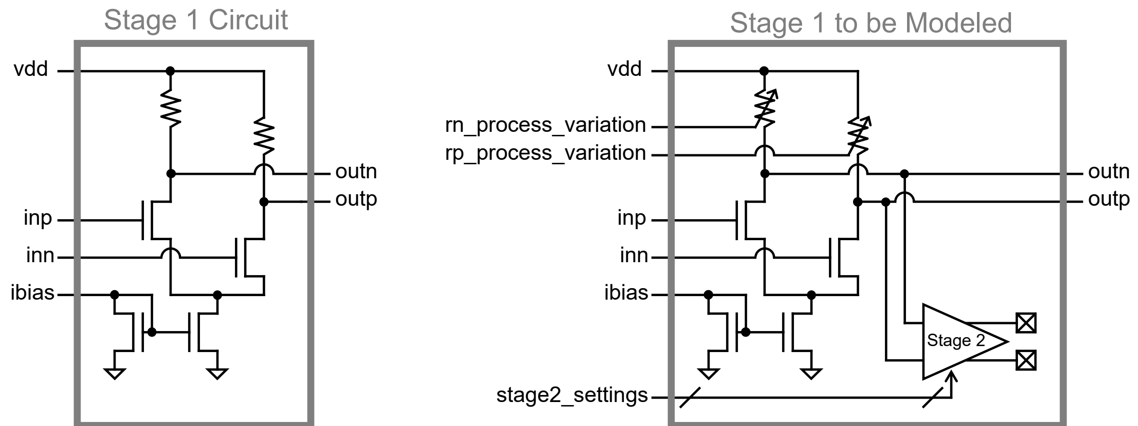


Figure 4.3: Example of a differential amplifier circuit with modifications to allow modeling of loading effects and process variation.

output corresponds to the output of the circuit under load, regardless of what is actually observing the outputs of the Verilog model. Instantiating the load in spice avoids the need for Fixture to have any extra complexity to define and instantiate loads itself. It is up to the user to pick a spice load model that balances accuracy, simulation time, and engineering effort. In many cases rather than determine a model of the load presented by the following circuit the user can simply instantiate the following circuit (or its first stage). This takes little effort and provides the most realistic load possible.

This method works well for static loads, but what about cases where the load might vary due to process variation, or even vary during the operation of the circuit (for example, if the following stage is a filter with adjustable passive components [48])? In these cases, the user is encouraged to model these effects in the wrapper like static effects, but give the wrapper additional inputs such as the bias current for the following circuit or a voltage that controls a variable capacitor as a proxy for process variation. Then, Fixture can use its existing functionality for handling optional inputs to characterize the circuit with respect to these additional inputs. An example of a wrapper including an adjustable following stage as well as process variation adjustment discussed in the next section is given in Figure 4.3.

4.3.6 Challenges with Process / Temperature variation

One benefit of the speed of analog functional models is the ability to run many tests over variations in process and temperature. Users can utilize Fixture as part of a system for characterizing their circuit with respect to process variation and efficiently determining its effect on system-level performance.

To model process variation the user has two options. One is to simply use a Monte Carlo extraction to produce multiple spice models, and then run Fixture separately on each one. The

resulting sets of extracted parameters give a a good way of measuring the sensitivity of different circuit behaviors to these Monte Carlo variations. The user can then calculate the variance of each parameter across these sets use this information to quickly build a large number of functional models with realistic variation, without needing to run a spice simulation for each one.

The second option for modeling process variation is to manually edit the spice model to add additional inputs that effectively control the values of important circuit components. Much like the strategy for dynamic loads in the previous section, this would allow Fixture to treat process variation as an optional input and model the circuit with respect to that variation. An example modeling the variation of two resistors is show in Figure 4.3. This strategy not only allows the user to produce functional models with realistic behavioral variation, but also shows the user the exact effect each component's variation has on each circuit parameter. This can help the analog circuit designer make informed decisions about how to improve the circuit's stability with respect to process variation.

Currently, *fault*, the library used by Fixture for testbench generation (Section 5.2), has no mechanism to adjust temperature mid-simulation since this is not a feature supported by many simulators. The only way to characterize with respect to temperature in Fixture is to run circuit characterization multiple times with different simulator arguments to change the temperature each run. This results in several models each corresponding to a different temperature, and the user can look at the parameters of each model to determine the circuit's variation due to temperature.

In the future we hope to have more automatic support for process variation built into Fixture. In the meantime users can create their own scripts to organize multiple runs of Fixture and produce the models they need for their use cases. Monte Carlo simulation is often a computational bottleneck and we believe that the functional models created with Fixture could be a solution.

4.4 Updating Equations with Optional Inputs

When the user defines optional inputs in the user configuration file, Fixture needs some way of allowing the modeled behavior to depend on these optional inputs. Since the modeled behavior is described by the parameter equations (Section 4.2), Fixture will modify those equations to take the optional inputs into account. In this section we will discuss the process Fixture uses to modify the parameter equations, and also the control the user has over how optional inputs affect circuit behavior.

4.4.1 Equation Hierarchy

Fixture's general approach is to replace each parameter in the equation by an expression involving the optional inputs. Since this can lead to more complex equations, when the modified equation is presented to the user, it is formatted as a hierarchy. This format allows pieces of the equation to be named and makes it easier for users to interpret the equation when there are a large number of

optional inputs.

To illustrate all the ways Fixture can extend an equation, we will consider a relatively complicated example extending a saturating amplifier template with two optional inputs: `vdd` and `r_adj[5:0]`.

$$\text{out} = \text{amplitude} \cdot f\left(\frac{\text{gain}}{\text{amplitude}} \cdot \text{in}\right) \tag{4.8a}$$

$$f(x) = \tanh(x) \tag{4.8b}$$

$$\text{amplitude} = \text{amplitude_nom} + \text{amplitude_vdd} + \text{amplitude_radj} \tag{4.8c}$$

$$\text{amplitude_nom} = A \tag{4.8d}$$

$$\text{amplitude_vdd} = B_1 \cdot \tilde{\text{vdd}} + B_2 \cdot \tilde{\text{vdd}}^2 \tag{4.8e}$$

$$\tilde{\text{vdd}} = \text{vdd} - \text{vdd_nom} \tag{4.8f}$$

$$\text{amplitude_radj} = g(\text{radj}) - g(\text{radj_nom}) \tag{4.8g}$$

$$g(\text{radj}) = \frac{1}{C_6 + \sum_{i=0}^{5} C_i \cdot \text{radj}[i]} \tag{4.8h}$$

$$\text{gain} = \text{gain_nom} + \text{gain_vdd} + \text{gain_radj} \tag{4.8i}$$

$$\text{gain_nom} = D \tag{4.8j}$$

$$\text{gain_vdd} = E_1 \cdot \tilde{\text{vdd}} + E_2 \cdot \tilde{\text{vdd}}^2 \tag{4.8k}$$

$$\tilde{\text{vdd}} = \text{vdd} - \text{vdd_nom} \tag{4.8l}$$

$$\text{gain_radj} = h(\text{radj}) - h(\text{radj_nom}) \tag{4.8m}$$

$$h(\text{radj}) = \frac{1}{F_6 + \sum_{i=0}^{5} F_i \cdot \text{radj}[i]} \tag{4.8n}$$

The first two equations, 4.8a-b, are written by the template writer. They are a function from `in` to `out` in terms of two parameters, `amplitude` and `gain`. They use a nonlinear function f, which is scaled by the gain and amplitude parameters, to model a saturating amplifier. In this case f is defined to be a `tanh` funciton, but it could be any function with slope 1 near the origin and saturation at ± 1.

The next block of equations, 4.8c-h, model the amplitude parameter as a function of the optional inputs. These equations are created by Fixture based on the settings in the user configuration file (see Section 4.4.2). Equation 4.8c defines the amplitude parameter as a sum of three influences: the nominal value, and additional effects due to the two optional inputs. It is important that the optional input effects are zero when the optional inputs are at their nominal values, and we can see how that is accomplished by looking at the definitions of those effects.

We can see that equation 4.8e is defined in terms of $\tilde{\text{vdd}}$, which is the deviation of `vdd` from its

nominal value. When `vdd` is at its nominal value then $\tilde{\text{vdd}} = 0$, and it is clear that $\text{amplitude_vdd} = 0$ as well. This centering around the nominal value of `vdd` allows the coefficients A and B_1 to have values with a more clear interpretation.[4] In general, Fixture will require all optional input effects to be zero when the optional inputs are at their nominal value. This makes it easy to separate different optional effects from each other and from the parameter's nominal value, which makes coefficients user-interpretable and easier to fit using regression (regression will be discussed in detail in Section 6.1).

We will skip over Equation 4.8g for a moment, and focus first on the definition of $g(\text{radj})$ in Equation 4.8h. This equation is nonlinear with respect to the coefficients for each bit of `radj`. It is a good model for the total resistance of a bank of resistors connected in parallel, which we introduced to model our motivating circuit in Equation 3.3.

Now we return to Equation 4.8g to see how Fixture guarantees that `amplitude_radj` is zero when `radj` is at its nominal value. We cannot follow the same approach as Equation 4.8e, which worked in terms of deviation from nominal `vdd`, because talking about deviation from nominal `radj` does not make sense when `radj` is treated in a bitwise fashion. When Fixture cannot center the optional input value directly, it falls back to the more general method of centering the entire effect of that optional input by subtracting one constant. In this case, the value of that constant is $g(\text{radj_nom})$.

Although it was not shown in this example, optional effects can also be functions of multiple optional inputs at once. In this case, it is still required that the value of the optional effect is zero when all optional inputs are at their nominal values.

In summary, Fixture models circuit behavior using a hierarchy of equations. At the top are equations created by the template writer that model an output in terms of an input and some parameters. Fixture can automatically create additional equations that model the parameters in terms of optional inputs. The equation modeling a parameter is always a sum of a constant nominal parameter value and optional input effects. The optional effects are always equal to zero when the optional inputs are at their nominal values.

4.4.2 User Configuration: Optional Input Dependence

Now that we have seen how optional input effects are used to define circuit behavior, we will discuss how Fixture creates the optional input effect equations. In general, Fixture will assume each parameter has a linear dependence unless the user specifies otherwise. In the user configuration, the user has the ability to define which parameters depend on which inputs and specify arbitrary dependence equations.

[4] To understand why centering affects A and B_1 but not B_2, imagine the result of expanding the expression $A + B_1(\text{vdd} - 3.0) + B_2(\text{vdd} - 3.0)^2$. With centering, the constant term A is simply the value of the amplitude at nominal `vdd`, but without centering the constant term becomes $A - 3.0 \cdot B_1 + 9.0 \cdot B_2$, which does not correspond to any physical circuit parameter. Similarly, with centering the linear term B_1 is the sensitivity of the amplitude to `vdd` at nominal `vdd`, but the non-centered linear term $B_1 - 6 \cdot B_2$ is not meaningful. The square term, B_2, is not affected by centering.

As an example, we will consider a linear differential amplifier model, which has 6 parameters. First, we will show the information in the user configuration file:

```
optional_input_dependence:
    gain_dc: []
    gain_cc: ['c0*vdd + c1*vdd**2']
    # offset_c has no entry, so it will use the default
    gain_cd: []
    gain_dd: [vdd, '1/(radj + c0)']
    offset_d: []
```

Next, we will show the set of equations that result from the template equations and this user configuration. We will not show the full hierarchy of equations here; instead we will make all the necessary substitutions to condense each parameter into one equation:

$$\text{out_cm} = \text{gain_dc} \cdot \text{in_diff} + \text{gain_cc} \cdot \text{in_cm} + \text{offset_c} \tag{4.9a}$$

$$\text{out_diff} = \text{gain_dd} \cdot \text{in_diff} + \text{gain_cd} \cdot \text{in_cm} + \text{offset_d} \tag{4.9b}$$

$$\text{gain_dc} = A \tag{4.9c}$$

$$\text{gain_cc} = B + C \cdot (\text{vdd} - \text{vdd_nom}) + D \cdot (\text{vdd} - \text{vdd_nom})^2 \tag{4.9d}$$

$$\begin{aligned} \text{offset_c} = & E + F \cdot (\text{vdd} - \text{vdd_nom}) \\ & + \sum_{i=0}^{5} G_i \cdot \text{radj}[i] - \sum_{i=0}^{5} G_i \cdot \text{radj_nom}[i] \\ & + \sum_{i=0}^{5} H_i \cdot \text{cadj}[i] - \sum_{i=0}^{5} H_i \cdot \text{cadj_nom}[i] \end{aligned} \tag{4.9e}$$

$$\text{gain_cd} = I \tag{4.9f}$$

$$\begin{aligned} \text{gain_dd} = & J + K \cdot (\text{vdd} - \text{vdd_nom}) \\ & + \frac{1}{L + \sum_{i=0}^{5} M_i \cdot \text{radj}[i]} - \frac{1}{L + \sum_{i=0}^{5} M_i \cdot \text{radj_nom}[i]} \end{aligned} \tag{4.9g}$$

$$\text{offset_d} = N \tag{4.9h}$$

We will walk through each of the lines in the user configuration to see its effect on the equations.

- `gain_dc`: This parameter is the gain from differential input to common mode output, and for a symmetrical amplifier its value should be zero. The user has listed its optional input dependencies as an empty list, so it has no dependencies. The corresponding line in the equations lists a constant to represent the nominal value, A.

- `gain_cc`: This is the common mode gain. Here the user has specified an arbitrary expression as a string, which is quadratic in `vdd`. The tool will recognize that it can replace `vdd` with its deviation from nominal in order to center the expression and ensure that it is zero when `vdd` is at its nominal value.
- `offset_c`: This is the common mode offset. Since the user did not specify how this parameter depends on optional inputs, the tool will model it as linear with respect to each optional input. Again it recognizes that `vdd` can be centered. For the two buses, it subtracts out the proper constant to make the value zero when the bus is nominal; the constant is a function of the bus's nominal value and the coefficients for each bit.
- `gain_cd`: This case is similar to `gain_dc`.
- `gain_dd`: This is the differential gain, which we know from Section 3.1.4 has a reciprocal relationship with the bits `radj`. The user has specified a default dependence on `vdd` as well as a custom expression for `radj`. Notice that where the user has simply written `'radj'` as part of the expression in the configuration, Fixture has substituted in a weighted sum of the bits, which is the default linear model for a bus.
- `offset_d`: This case is similar to `gain_dc`.

The user's custom expressions are parsed using the Sympy library which includes a large number of built-in functions including trigonometric functions, piecewise functions, and more [49]. The heirarchy of Sympy expressions can then be combined together and evaluated using numpy [50] for efficient evaluation when doing parameter fitting.

Keeping with our guiding principle of smart defaults but advanced options, we expect that the default linear case will be suitable for most circuits, but allow the user to specify any function. In most cases linear dependencies are a good first pass and the user can then modify the dependencies to remove them or make them more sophisticated as needed.

4.5 Extending a Template with Vectored Inputs and Outputs

Besides adding optional inputs, the other way the user can change the physical pinout of the template is with vectoring. Vectoring takes a required input or output and splits it into multiple physical pins. By far the most common usage is to take a single-ended circuit and turn it into a differential circuit. Vectoring is also useful, for example, to add a calibration version of an input. Note that a calibration input often cannot be added as an optional input because it doesn't just affect the behavior of the main input; it completely replaces the main input in certain operating modes.

In this section we will discuss how Fixture vectors the parameter equations, and in Section 5.3.2 we will discuss how Fixture vectors the testbench. The process of vectoring a parameter equation

can be broken into two steps. The first is determining which measured data will be vectored, and the second is vectoring the equations themselves.

When determining which measured data should be vectored, it is important to remember that some of the inputs to a parameter equation are not the circuit's physical inputs and outputs directly, but rather processed versions of those inputs and outputs. The user only reports which physical inputs and outputs have been vectored, and it is up to the template writer to know how those will affect the processed inputs and outputs. Consider the linear amplifier equation:

$$\text{out} = \text{gain} \cdot \text{in} + \text{offset} \tag{4.10}$$

It is clear that the term `in` should be vectored when the physical input is vectored, and the term `out` should be vectored when the physical output is vectored. But the approach is not as clear when the terms in the parameter equation do not correspond directly to the physical inputs and outputs. Consider the equation for a pole location in a dynamic amplifier:

$$\text{p1_measured} = \text{p1} \tag{4.11}$$

Here, `p1_measured` is a processed output, meaning the values were calculated in the Analysis section of the testbench. The testbench writer knows that this value was obtained by processing a waveform of the output of the amplifier, but from the tool's perspective the Analysis code is a black box and it cannot tell which circuit pins were used in the creation of `p1_measured`. Because of this, the tool cannot automatically determine that `p1_measured` should be vectored exactly when the amplifier output is measured. Rather than have the tool inspect the Analysis code to try to determine the origin of `p1_measured`, we simply ask the template writer to report which processed outputs are functions of which circuit pins for the purpose of vectoring.

Another way to illustrate the need for template writers to tell Fixture which processed outputs correspond to which circuit pins is to look at a case where the template writer could have chosen two different implementations. In our dynamic amplifier example the template writer could have chosen to place the filter before the amplification rather than after. Doing so would require a different pole/zero extraction strategy in the case of a differential amplifier, so there are two different, but both reasonable, ways of implementing this template. In the case without any vectoring and a linear amplifier, the behavior of these two versions is identical, so Fixture will not be able to determine automatically which version was chosen. The different versions are illustrated in Figure 4.4.

The tool decides which parameter equation inputs and outputs to vector based on which physical pins are being vectored and the template writer's specification of which inputs and outputs depend on which physical pins. Then, when the simulation is done and the processed inputs and outputs are produced, the tool checks that the vectored processed inputs and outputs match the ones the template writer specified. More information about how the processed inputs and outputs are produced

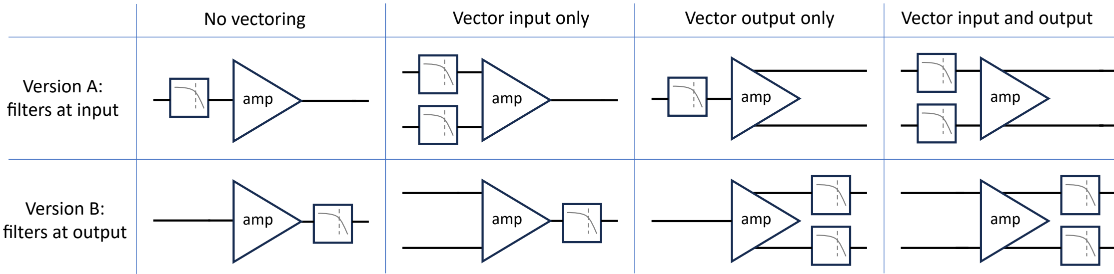


Figure 4.4: When vectoring an amplifier model with dynamic behavior, the decision whether to vector the filter depends on whether the filter comes before or after the static amplification. In Version A the filter comes before the amplification, so the vectoring of the filter is linked to the vectoring of the input. In Version B it comes after, and its vectoring is linked to the output. In the case where neither or where both input and output are vectored, versions A and B have the same expressive power as long as the filter and amp are linear.[5] It is up to the template designer whether to implement Version A or B, but in the middle two columns their choice determines whether the filter needs to be vectored, so they need to explicitly tell Fixture whether the filter is linked to the input or the output.

is given in Section 5.3.

The second step of vectoring the parameter equations is updating the equations themselves. We can begin with a simple example. We will take a linear amplifier equation with gain and offset, and vector the input:

$$\text{out} = \text{gain} \cdot \text{in} + \text{offset} \tag{4.12a}$$

$$\text{out} = \text{gain}_0 \cdot \text{in}_0 + \text{gain}_1 \cdot \text{in}_1 + \text{offset} \tag{4.12b}$$

The user requested that the physical input be vectored into two pieces, and the tool knows that this vectors the measured data for `in` into the components [$\texttt{in}_0$, $\texttt{in}_1$]. We see that in our resulting equation, the **`gain`** parameter has also become vectored. In cases where the vectored input is multiplied by a parameter, Fixture will vector the parameter as it does in Equation 4.12. Fixture extends this strategy to cases where the coefficient is not direclty multiplied by the input by searching for a coefficient that can be distributed next to the input. This is the case for Equation 4.4, a nonlinear amplifier model using tanh to model gain compression. That equation is reproduced here, along with a vectored version:

[5]For the two-input, two-output case we can see that these models are equivalent by representing each filter with an s-domain transfer function and the amplification with a 2x2 matrix. We can place the two transfer functions on the diagonal of a 2x2 diagonal matrix X, and represent linear amplification in the 2x2 matrix Y. If Version A applies XY, Version B can equivalently apply $Y(Y^{-1}XY)$. When Y is singular, the pseudoinverse can be used instead.

$$\text{out} = \frac{\text{v_max} - \text{v_min}}{2} \cdot \tanh\left(\frac{2 \cdot \text{gain}}{\text{v_max} - \text{v_min}} \cdot (\text{in} - \text{in_offset})\right) + \frac{\text{v_max} + \text{v_min}}{2} \tag{4.13a}$$

$$\text{out} = \frac{\text{v_max} - \text{v_min}}{2} \cdot \tanh\left(\frac{2}{\text{v_max} - \text{v_min}} \cdot (\text{gain}_0 \cdot \text{in}_0 + \text{gain}_1 \cdot \text{in}_1 - \text{gain_times_in_offset})\right) + \frac{\text{v_max} + \text{v_min}}{2} \tag{4.13b}$$

Although there is no parameter multiplied by `in` directly, the tool sees that it is possible to distribute `gain` inside the parenthesis so that it is directly multiplied with `in`. As a result Fixture also needs to convert the `in_offset` parameter into `gain_times_in_offset`. Most of the time we find that this produces the named parameters the user is expecting. Unfortunately, some equations have a form such that Fixture cannot find a coefficient to multiply into the vectored input:

$$\text{out} = A \cdot \text{in}^2 + B \cdot \text{in} + C \tag{4.14a}$$

$$\text{out} = A \cdot (\alpha \cdot \text{in}_1 + \beta \cdot \text{in}_2)^2 + (B_1 \cdot \text{in}_1 + B_2 \cdot \text{in}_2) + C \tag{4.14b}$$

The B term is able to vector its parameter as in the previous examples, but the A term cannot do this because the input is squared. Instead, Fixture uses its fallback strategy of replacing the vectored input with a linear combination of its components. Another way to understand this is that Fixture is projecting the 2-dimensional input space into 1 dimension along an axis defined by α and β so that it can use the expression which was originally defined over 1 dimension. Fixture will also constrain α and β such that the sum of their magnitudes is 1 to remove an unnecessary scaling factor.

4.6 Extending a Template with Custom Domain Changes

Custom domain changes allow us to support a wide variety of user circuits while keeping the template library small. Functionally, the custom domain change is a translation from one domain to another at the physical pinout level (Figure 4.5). If the user circuit has an input in a special domain, they can create a proxy signal which is defined as some function of the physical input. Then, when the template needs to use the value of that proxy input it can apply the custom function to find the value it needs.

The translations are handled by Fixture, and the template writer will not interact with any data until it has gone through this translation. We intend for contributors to be able to create their own custom domain changes if needed. Like template writing, creating a new type of custom domain change would require some knowledge of python and the tool's internal systems, so we provide some pre-written custom domain changes that can be easily instantiated by the user. We organize these

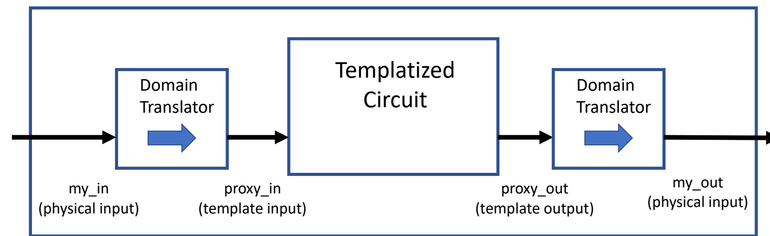


Figure 4.5: If the user circuit communicates inputs and outputs in a special domain, these inputs and outputs need to be translated to work with the template. The user can create proxy signals in the template's expected domain, and specify the domain translations needed to go between physical signals and their corresponding proxy signals.

pre-made domain changes into two classes: linear transformations, and time-based transformations.

4.6.1 Linear Transformations

Linear transformations allow for one signal to be a linear combination of other signals. This transformation is used often to convert between a positive / negative domain and a differential / common mode domain. At the input, a proxy signal is created to represent the differential input and is specified as the difference between two physical inputs. Then, a second proxy input is created to represent the common mode input and specified as the average of the two physical inputs. The user can specify any non-singular linear transformation, so less-common definitions of differential input, such as the circulating current discussed in the motivating circuit (Section 3.1.1), are also possible. A multi-wire link with more complicated modes than just differential and common mode can also be represented with these translations as long as the modes are linear functions of the physical inputs.

Here is a verbose example of how the user can specify a transformation of circuit inputs from the positive / negative space to the differential / common mode space:

```
proxy_signals:
    indiff:
        style: linear_combination_in
        components: [inp, inn]
        coefficients: [1.0, -1.0]
    incm:
        style: linear_combination_in
        components: [inp, inn]
        coefficients: [0.5, 0.5]
    input_vector:
        style: vector
        components: [indiff, incm]
```

```
template_pins:
    input: input_vector
```

From the template's point of view, the differential and common mode proxy inputs are the only inputs, and the template designer does not interact with the positive and negative physical inputs at all (refer again to Figure 4.5). When creating testbench stimulus the template writer works in the differential / common mode domain only, and Fixture automatically translates that stimulus to the positive / negative physical domain before creating the fault testbench (See section 5.2 for more information about fault testbenches).

4.6.2 Time-Based Transformations

If the template expects a particular output to be in the voltage domain but the user specifies that it should be interpreted in the time domain, Fixture will use time-domain reads to do the conversion automatically. For example, when working with the DragonPHY project [51, 52] we had a sample-and-hold circuit that gave its output as a pulse-width rather than a voltage (Figure 4.6). In the user configuration file we can define the output pulse as a function of the physical output voltage:

```
proxy_signals:
    output_pulse:
        style: pulse_width
        reference: output
```

Now we can use the name `output_pulse` to refer to the width of the pulse elsewhere in the user configuration. When mapping template inputs to user circuit inputs we can map this pulse width to the sample-and-hold template's output:

```
template_pins:
    sampler_input: input
    sampler_clk: clk
    sampler_output: output_pulse
```

It is important to note when the read occurs relative to the time of the pulse. The template writer expects to be reading a voltage, and will wait a user-specified amount of time after the sample-and-hold takes its sample before reading. In the pulse-width circuit, the pulse begins immediately after the clock edge, so the read does not take place until after the pulse has finished. For this reason, the default way a pulse width is read by Fixture is to look backwards in time from when the `get_value` function is called. More information about how the proxy signal is implemented and about how the pulse width is extracted from the waveform is given in Section 5.3.3.

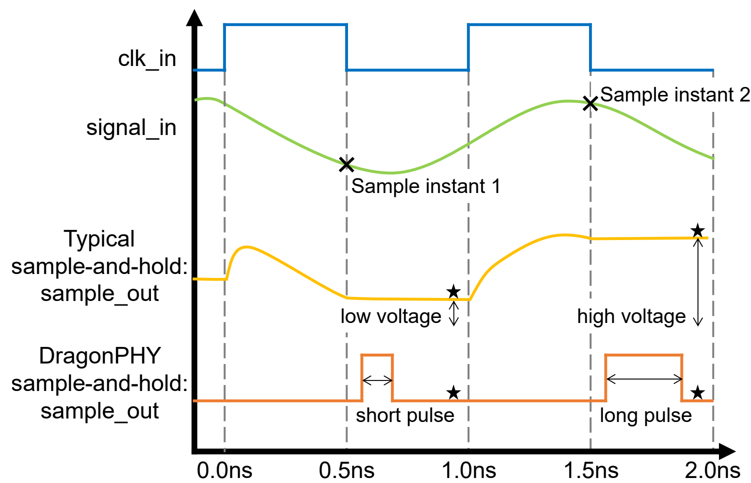


Figure 4.6: Comparison between a typical sample-and-hold output and the pulse width output from the DragonPHY project's sample-and-hold. In a typical circuit the output tracks the input signal when the clock is high (0.0-0.5ns), then holds the value at the instant the clock falls (samples at 0.5ns and holds from 0.5ns to 1.0ns). In Fixture's sample-and-hold template, the held value is measured at the time marked by the star (just before 1.0ns) to allow the held value time to settle before measurement. In the case of the DragonPHY sampler, the output is digital. Shortly after the clock falls there is a pulse on the output with a pulse width proportional to the sampled voltage. The time that the Fixture template measures the output, marked with a star, is after the pulse finishes so the template must search backwards to find the pulse.

4.7 Using the Template

Throughout this chapter we have described what the template is and the many ways it can be modified. At its core, the template has equations describing the circuit to be modeled and instructions for how to create a corresponding testbench. These have both been extended according to the specific needs of the user's circuit, including optional inputs and custom domain transformations. In the next chapter we will focus on the details of how the testbench instructions become an actual spice or Verilog testbench, with the goal of providing data for model fitting in Chapter 6.

Chapter 5

Testbench Generation

Generating a good testbench is an essential part of characterizing a circuit. The template library is an excellent way to capture and reuse good strategies that are circuit-specific. The ultimate goal of Fixture is to ensure that any circuit effect an analog designer can think of can be measured automatically with the proper template. If the analog designer knows about a circuit behavior then they can probably write a spice testbench that causes the behavior to occur, and Fixture just needs to make sure it's possible to include that testbench in the template. The challenge is that Fixture then has to extend that testbench to match the user's specific circuit including optional inputs, vectored inputs, user-specified domain transformations, and user circuits modeled in spice or Verilog. To accomplish this Fixture relies on the testbench generation library *fault* and some guidelines for how the template writer should write the testbench. In this chapter we will discuss how Fixture chooses sample points to ensure good coverage of the circuit's input space, the testbench generation library *fault* including the features we added specifically to support Fixture, and the process a template writer must follow to write the testbench section of the template.

5.1 Choosing Input Points

In this section, we will discuss part of the strategy used to choose pseudo-random inputs for circuit characterization. Inputs are carefully chosen to fully exercise the space of circuit behaviors, make model fitting easy, and produce useful plots for the user. Here we will only discuss the first point, exercising the input space, and save discussion of model fitting and plotting for Section 6.1.4. The goal of this section is to describe a random sampling primitive: choosing a fixed number of sample points over the space spanned by a set of input signals. This primitive will then be used by the model fitting code to choose the full set of input points.

5.1.1 Latin Hypercube Sampling and Orthogonal Sampling

The reason Fixture uses random input points rather than, for example, evenly spaced input points is to avoid aliasing with other effects. Most commonly, issues occur when a particular input has multiple batches of samples, because if each batch is chosen the same way then samples are duplicated between the different batches. Duplicated samples give less information about the circuit's behavior per simulation time, so they should be avoided if possible. For example, if Fixture is choosing 100 sample points for the differential and common mode inputs, it is not optimal to choose 10 values for the common mode, and then at each of those common mode points choose the same batch of 10 differential inputs. Although this is 100 different sample points, it only includes 10 unique values for the common mode and 10 for the differential, which does not gather as much information as possible. It is also possible that a circuit's response is periodic with respect to an analog input, and in that case sampling evenly-spaced points could alias with the periodic response and mask some circuit behavior.

Using pseudorandom samples avoids these negative effects; however, choosing input points uniformly at random does not guarantee even coverage of the input space. Latin Hypercube Sampling (LHS) and Orthogonal Sampling are two techniques used to constrain a set of random samples to guarantee certain coverage properties. These techniques are widely used for Monte-Carlo sampling [53, 54]. We will briefly discuss the guarantees made by these techniques for N sample points spanning M input axes. Figure 5.1 gives visual examples for 30 sample points over 2 input axes. LHS guarantees that if an axis is broken into N equal pieces, each piece will contain exactly one sample. This property holds independently for each of the M axes. Orthogonal Sampling, as used by Fixture, breaks each axis into $K = \lfloor N^{1/M} \rfloor$ equal pieces. We can then define K^M subspaces by choosing one piece from each axis, and we guarantee that each of these subspaces contains at least one sample point.

In the context of circuit analysis, we can immediately see the positive effect of each of these techniques. Imagine a circuit with inputs A and B, where A affects the output significantly and B has very little effect. LHS guarantees that the sample points are well-distributed to capture the effects of A on the output. Next, imagine a circuit with inputs C and D, with a single output that increases as C increases and also increases as D increases. Orthogonal sampling guarantees that there are some samples where both A and B are both high, nearing the highest possible value of the output, and some samples where A and B are both low, nearing the lowest possible output.

5.1.2 Scaling Input Samples

Section 5.1 explained how Fixture creates pseudorandom samples along an axis. Before those can be applied to inputs they must be scaled to the proper range. For most analog inputs this is extremely simple: from the user configuration the tool knows the maximum and minimum allowable value for each input, so the samples are scaled linearly to fit in that range. For quantized analog inputs, the

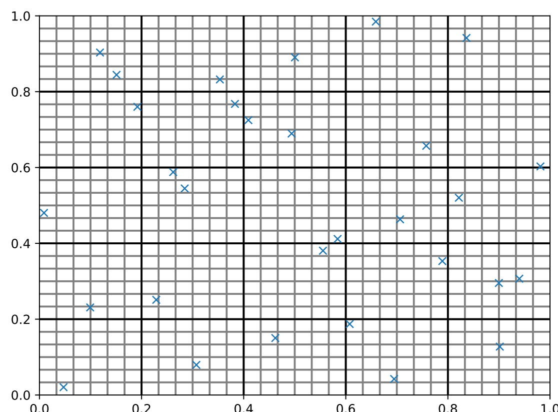


Figure 5.1: 30 sample points chosen across two input dimensions using Latin Hypercube Sampling and Orthogonal Sampling. This one set of sample points satisfies both constraints simultaneously. LHS guarantees exactly one sample in each row and column of the 30x30 grid. Orthogonal sampling guarantees at least one sample in each cell of the larger 5x5 grid.

process is similar. We ask the user to provide information about the style of the binary bus, for example whether it is binary-coded or thermometer-coded (Section 4.3.3), so we are able to convert between input codes and decimal numbers. To convert the pseudorandom sample to a decimal input we simply scale it to the decimal range and round to the nearest whole number, taking care with the endpoints to ensure that they have the same probability of occurring as every other value.

There are other reasonable ways to sample quantized analog inputs that are not used by Fixture. DaVE, for example, relies on the fact that the effect of each bit adds linearly with the effect of each other bit. The random sampling therefore ensures that each bit is equally likely to be sampled as a zero or a one, without considering the decimal value of the bus. We will call Fixture's approach the decimal method, and DaVE's approach the binary method. In the case of a thermometer code, we prefer the decimal method for two reasons. The first is that the binary method is unlikely to produce inputs where all the bits are simultaneously zero or simultaneously one. This means that the extreme values of the input are rarely exercised, which can hide nonlinear behavior in the circuit (Figure 5.2). The second issue with the binary approach is that it often produces combinations of input bits that would never be produced by the digital circuitry driving the user circuit in practice. Specifically, most digital circuitry that creates thermometer codes will always place the ones in the low-order or the high-order positions. In other words, a value of 4 in an 8-bit bus usually reads "00001111" or "11110000", but never reads something like "00101101". We would like the inputs produced by Fixture to match the inputs the user circuit would see in practice, and by asking the user for the necessary information about bus the decimal method produces inputs that meet that requirement. It is important to note that the decimal method used by Fixture also has some drawbacks. It does not guarantee that each bit receives an equal number of zeros and ones, as the binary method can

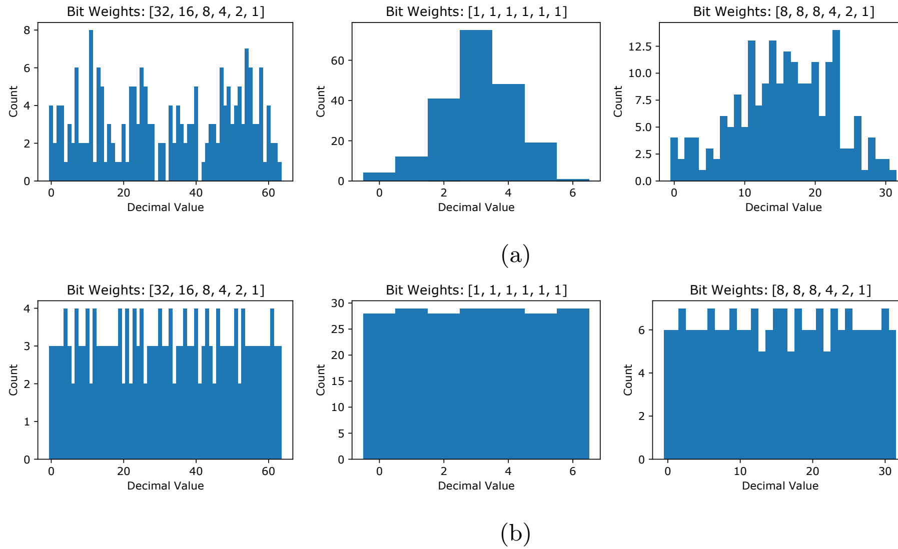


Figure 5.2: Histograms showing how many times each decimal value is chosen out of 200 samples. (a) Using the binary sampling strategy, where each bit is sampled independently. On the left, the binary weighted bus has samples randomly distributed, which may miss a few possible bus values. In the middle, the thermometer-coded bus is sampled very poorly because the extreme values are very unlikely. On the right, the segmented bus is somewhere in between. (b) The same histograms, but using the decimal sampling strategy. In this case, Latin Hypercube Sampling[1]ensures that each decimal value is chosen the same number of times, plus or minus one. These flat histograms offer much better coverage of all circuit behaviors than the ones in Figure a.

easily guarantee. In practice, the decimal method will produce a random mix of zeros and ones for each bit which is sufficient with the sample sizes used by Fixture.

5.1.3 Custom Input Constraints

In Section 3.1.2, we described a user circuit with two optional inputs whose values need to obey a certain relationship. Specifically, the `radj` and `cadj` buses always need to be chosen such that the time constant set by the feedback resistor and capacitor is within a certain range. It is not acceptable to sample them each independently because that would create some disallowed input pairs. In order to generate input stimuli for this circuit we need to break our usual convention of treating each optional input as a separate input dimension. To allow the user to do this we created custom input constraints.

Normally when specifying input stimuli in the user configuration, the user maps each input to an allowable range. In the case when two optional inputs need to be sampled together, the user can instead map a tuple of inputs to a dictionary with more information about the allowable samples.

[1]Fixture uses both LHS and Orthogonal Sampling when choosing the decimal values, but because these plots only have one dimension (the bus value) Orthogonal Sampling has no effect.

The dictionary contains the range for each individual input dimension, and also a constraint function that every sample must obey. In the motivating circuit example we had an adjustable resistance and adjustable capacitance, and their product had to stay within a certain range. We can accomplish this by creating a constraint function that calculates their product and returns `True` only if it is within the acceptable range:

```
stimulus_generation:
indiff: (-3e-6, 3e-6)
incm: (-1e-6, 1e-6)
vdd: (2.7, 3.0, 3.1)
(cadj, radj):
    cadj: (0, 63)
    radj: (0, 63)
    filter_fun: `1000 < cadj*radj < 2000'
```

We can see that `indiff`, `incm`, and `vdd` specify their ranges with tuples (`vdd` also includes a nominal value as the middle of three tuple entries). `cadj` and `radj` instead use the constraint dictionary.

Functionally, Fixture uses rejection sampling to create these constrained samples. It simply generates a random sample within the available ranges for each of the input dimensions and will throw that sample away if it does not follow the constraint. In most cases, the computational overhead of generating and then discarding most samples is negligible compared to the simulation and model fitting done by Fixture.

A different approach to solve this problem is to ask the user to provide a function to map the full space of sample points to the space of acceptable sample points. This method has a computational advantage in the case where the size of the acceptable space is a very small fraction of the total space. Depending on how the user writes the mapping function it could also allow these inputs to follow the orthogonal sampling constraints with respect to the other input dimensions. However, we found the mapping functions to be more difficult to write than the filter functions, and therefore did not take this approach.

5.2 The Testbench Description Language *fault*

An essential part of Fixture is the ability to compile testbenches that have been customized to the user's specific circuit. To accomplish this, Fixture utilizes a tool called *fault* [55]. Additionally, we would like to be able to characterize both spice circuits and existing Verilog functional models with the same template; *fault* allows us to use a single testbench to exercise both of these model types. In this section we will discuss the advantages of using *fault*, as well as the features that we added to *fault* specifically to support Fixture.

5.2.1 Testbenches Written in Python

The main advantage of *fault* is the ability to write testbenches in Python and then compile them to spice or Verilog. This is great for Fixture because we need to tailor the testbench to match the user's specific instance of a circuit, and the flexibility of python allows us to do that. Fixture's approach to is to have the template writer create a short testbench to collect one datapoint in *fault*, and then automatically create a full testbench from that information. More information on how the template writer creates the single-datapoint testbench is given in Section 5.3.1.

The fact that *fault* testbenches are written directly in python makes it easy for Fixture to use advanced python features and existing libraries to prepare the testbench [56]. For example, Fixture needs to interact with the filesystem to read the user configuration file that gives information about the user's specific circuit. Additionally, Fixture can use existing python libraries to parse the user's spice or Verilog circuit and check that the pinout matches what the user specified in the configuration file. Any amount of pre-processing that Fixture needs to do is easy to integrate with the testbench because they are both part of the same python program.

Within the testbench description itself, the testbench writer can continue to rely on python features, including some amount of flow control. For example, when writing the sample-and-hold template, there was a need to structure the ramp input differently depending on the specifics of the sample point being simulated. For the input slope test, Fixture randomly chooses an input slope between a large positive and large negative value. Because the slopes are typically steep, Fixture ramps the input from the minimum to the maximum possible voltage (or vice versa for negative slopes) to give the circuit as much time as possible to settle during the ramp. Occasionally, however, the randomly-chosen slope will be very close to zero and in those cases it would take too long to ramp all the way from the minimum to the maximum possible input value. In this case the template writer needs to limit the ramp start and stop points based on a maximum time rather than maximum and minimum input voltages. Luckily, it is easy to detect and handle these small-slope cases separately using python's flow control. With *fault*, the tool can wait until testbench compile time, when it already has information about the user's circuit and the random samples, to choose which branch of the flow control to take.

Because the template writer's testbench only describes the measurement of one datapoint, Fixture has to iterate that short testbench many times with different random values and different settings for any optional inputs. With *fault*, instantiating another copy of the single-datapoint testbench is as easy as making another call to the function that defines it. Optional input values can be set by Fixture before that call is made, and any information the template writer needs can be passed as arguments to that call.

Although some of the features described in this section are already available in existing spice and Verilog simulators, spice and Verilog have different syntax and so it is convenient to use *fault* to write just one testbench that can be compiled to a variety of simulators. Additionally, different

simulators for the same language sometimes have different compatibility with various spice models or nonsynthesizeable Verilog features like Direct Programming Interface (DPI) calls. DaVE's solution to this issue was to use a single mixed-signal simulator, Cadence's Spectre AMS, for everything. As mentioned earlier, this solution is not ideal, however, because that software is proprietary and not all circuit models are compatible with it. Luckily, with *fault* switching simulators is as easy as updating the list of simulator command-line arguments, which allowed us to use ngspice [57] with Skywater-130 open-source PDK design, since it was the only simulator where these models worked well.

5.2.2 Domain Translation

As we saw in Section 2.4.2, it is common for engineers to think about their circuits in domains other than the physical domain. In order to facilitate the writing of testbenches for these circuits we allow template writers to interact with the circuit in the domain they prefer and then Fixture handles translation to the physical domain automatically. In this section we will discuss three different classes of transformations that the tool can do automatically.

Linear Transformations

The simplest type of domain translation is a linear transformation at the inputs or outputs of a circuit. Fixture ensures that the input and output ports, as well as variables holding values corresponding to those ports, are of the correct types so that when the testbench writer makes calls to the "poke" and "get_value" functions to set and read port values Fixture will compile the testbench with physical values in the correct domain. This feature is especially useful since the template writer needs to write a testbench without knowing whether a port is physically one signal or a pair of differential signals. With this setup, they can use the exact same line of code to interact with the user's circuit whether the variables refer to a single value or a vector of values.

Here is an example of an extremely simple `testbench` function and the corresponding `analysis` function. The exact same functions can be used to characterize a single-ended or differential amplifier. The difference is that in the differential case, the values of the variables are all vectors in the differential / common-mode space.

```
def testbench(self, samples):
    # set up variables
    input_port = self.ports['input']
    output_port = self.ports['output']
    delay = self.test_options['delay']
    input_sample = samples['input']
```

```
    # define testbench
    fault.poke(input_port, input_sample)
    fault.delay(delay)
    out = fault.get_value(output_port)
    return {'out': out}

def analysis(self, results):
    out_value = results['out'].value
    return {'out': out_value}

# Example variable values for a single-ended amplifier
input_port = dut.input
output_port = dut.output
input_sample = 0.8
output_value = 1.1

# Example variable values for a differential amplifier
input_port = [dut.input_diff, dut.input_cm]
output_port = [dut.output_diff, dut.output_cm]
input_sample = [0.8, 0.2]
output_value = [1.1, 0.21]
```

To implement this, we took advantage of dynamic typing in python. We changed the `poke` function to detect when the input port and value are vectored, do the appropriate linear transformation, and poke the individual ports. Note that the user provides the linear transformation from the physical space to the conceptual space, so Fixture actually applies the inverse transform on the inputs. Then, after the testbench has been run but before the `analysis` method we read the values of each individual port and apply the appropriate linear transformation, placing the results in `results['out'].value`. We believe it would be difficult to provide a similarly simple experience to the testbench writer in Verilog or spice directly.

Writing Time-Based Signals

When designing circuit testbenches it is common to work with signals that are defined in the time domain. For example, the phase blender we considered in Section 2.4.2 has two inputs which are clocks at a specific frequency and relative phase. Additionally, a sample-and-hold circuit has one, or sometimes multiple, clocks that need to run to control the sampling. While it was always possible to exercise these inputs in *fault* by scheduling each edge in order, we added new functionality to *fault* to manage these time-based signals automatically.

The new *fault* `background_poke` feature allows the testbench writer to poke an input with a time-based value such that the effect does not happen immediately, but happens on a schedule defined by the type of `background_poke`. Currently there are four types: clock, sine, ramp, and future. Clock simply defines a digital clock with a user-specified frequency and phase, and will continue to drive that clock until the node is driven to a different value. Sine is very similar, driving a sine wave with a user-defined frequency, amplitude, offset, phase, and error tolerance. In transient spice sims inputs are piecewise-linear, and in Verilog they are piecewise-constant, so we approximate the sine wave by scheduling many changes according to the error-tolerance set by template writer. The ramp type is used to define a slowly ramping input, and accepts a slope, stopping point, and error tolerance. Finally, the future type allows the user to schedule a list of arbitrary input changes to happen in the future. This is used in the sample-and-hold template to schedule clock inputs with specific jitter characteristics.

To implement this functionality, we created a pool of `background_poke` objects that exists while *fault* is compiling the testbench. When the user makes a new `background_poke` it gets added to the pool, and when a user pokes the same input again it gets removed from the pool. Any time a delay happens *fault* checks the pool for edges that should occur during that delay, using a heap to efficiently organize the upcoming event queue [58].

Reading Time-Based Signals

Like we did for writing time-based signals, we also extended *fault* to be able to read time-based signals. For example, we may need to read the frequency of the output clock from a voltage-controlled oscillator. Before our changes, *fault* had no way to do this besides reading the output voltage at high frequency and noting when the output changed. With the new strategy, *fault* can directly read the output waveform for the exact time of the edges.

Our `domain_read` function allows the user to read several different types of time-based signal. Currently, the supported types of signals are edge, frequency, pulse_width, and block. By default, each of these functions looks backwards from the specified time to find the most recent edges needed to define the requested value. Edge is the simplest, and returns the time that an edge occurred within the window specified by the user. Frequency measures the frequency of an output clock based on the time between two consecutive rising edges. Pulse_width is similar, finding the time between a rising edge and the following falling edge. Block simply returns a portion of the waveform over a user-specified interval, allowing the user to do their own post-processing.

One challenge with the implementation of these functions is that reading the output waveform must be done differently for spice and Verilog. Both waveforms can be read as [time, voltage] pairs, but spice results are interpreted as piecewise-linear while Verilog results are interpreted as piecewise-constant. In order to implement each of these functions without duplicating code, we wrote an edge-finding primitive for spice and Verilog and then built each function out of calls to the

edge-finding primitive (Figure 5.3).

The edge-finding primitive takes as input a starting time, a direction to search, whether to find a rising or falling edge, and a voltage level to slice at. It operates by looking at the waveform at the specified time, and then stepping through each [time, voltage] pair in the specified direction until the waveform crosses the slicing level in the specified rising or falling direction.[2] For a Verilog waveform the time of an edge is simply the time of the datapoint that caused the voltage to cross the slicing level. For a spice waveform, the tool considers the datapoints on either side of the crossing and uses them to determine the exact time the slicing level was crossed.

The different read styles (except for the block read) can each be implemented with one or more calls to this edge-finding primitive. Additionally, the template writer can use it to build different kinds of reads outside the ones provided automatically. For example, if the template writer wants to know the rise time of a particular edge they can do two calls to the edge-finding primitive with the slice level set to 10% and 90% and then take the difference of the results.

In addition to the spice and Verilog versions of the edge-finding primitive, we implemented a third version for mLingua signals. As discussed in Section 2.4.3, mLingua is a Verilog library, but rather than express signals as piecewise-constant it uses an explicit piecewise-linear representation. Each datapoint is a [time, voltage, slope] tuple. Implementing the edge-finding primitive with this information is straightforward, and after implementing this one function the different read types are all available for use with mLingua.

In addition to being used by the template writer, the domain read functions can also be used by the end user. Recall that in Section 4.6.2 we saw an example of a sample-and-hold circuit with a pulse-width output. The user was able to define a proxy signal for the width of the pulse and tell the template to treat that as the sampler output:

```
proxy_signals:
    output_pulse:
        style: pulse_width
        reference: output
template_pins:
    sampler_input: input
    sampler_clk: clk
    sampler_output: output_pulse
```

Inside the sample-and-hold template, the code that reads the output from the sample-and-hold circuit uses *fault*'s `get_value` function to read the output:

[2]For spice results, the tool needs to be careful about its starting point: in general it needs to look back at the last datapoint before the start time, but should be careful not to include crossings before the start time. For example, a crossing may happen at 0.5ns during the linear piece defined by endpoints at 0ns and 1ns. If the start time is 0.25ns, the tool needs to look backwards to the 0ns datapoint to find the crossing at 0.5ns. If the start time is 0.75ns the tool will still look backwards and find the 0.5ns crossing, but needs to discard it because it is before the start time.

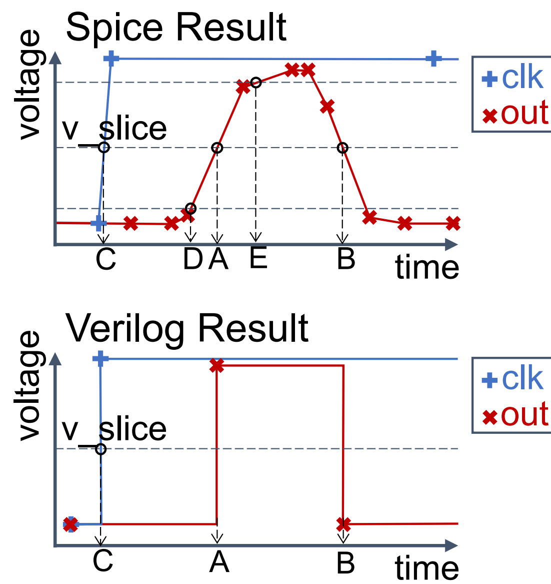


Figure 5.3: Measurement of a pulse width using the edge-finding primitive. For both the spice and Verilog results, the primitive finds the times of the rising and falling edges at A and B, and their difference is the pulse width. The template writer can use the primitive to find additional information about the pulse, for example, the time that the pulse occurs relative to the clock edge is $A - C$. In the spice result, the template writer can also measure the rise time of the edge by changing the slice level and finding the 10% and 90% crossing times D and E.

```
out = fault_tester.get_value(sampler_output)
```

Based on the user configuration, Fixture has automatically set the `sampler_output` to have the correct type to be read as a pulse width, so the `get_value` works correctly without any changes to the template.

5.3 Templatized Testbenches

The core of Fixture's analysis template is the templatized testbench. This testbench is a small slice of a typical analysis testbench which will be extended, repeated, and translated by Fixture into the final testbench run by the simulator. This section focuses on the code written by the template writer, with the goal of making clear what testbenches are and are not possible with Fixture.

5.3.1 Writing a Templatized Testbench

Recall that a template is built from a collection of tests, and each test typically focuses on one aspect of the circuit. The testbench described in this section is one piece of a test, describing how to run a simulation to collect information on a particular aspect of the circuit (recall Figure 4.1). The testbench writer has almost complete freedom to write any *fault* program in order to exercise the circuit. In this section we will first discuss the inputs that the template writer has to work with, and then discuss the limitations that the template writer should keep in mind when writing the testbench.

The testbench writer is provided with references to all the required template pins. This means that they have full freedom to set and read those pins, but have no access to the optional inputs (Exceptions to this are discussed in Section 5.3.3).

The testbench writer can also request and is provided with any pseudorandom numbers needed for the testbench. One example is the size of a step when measuring an amplifier step response. We discourage the template writer from using a random number generator in their code because we want to follow the random constraints described in Section 5.1.

Finally, the template writer can use additional information about the circuit provided by the user. We allow the template writer to request an arbitrary python dictionary from the user (specified in the user configuration file) with any additional information needed to run the testbench. We call this dictionary the "test options" dictionary. For an amplifier, the test options dictionary simply contains a value for the approximate settling time of the amplifier. While it is possible to write a testbench to figure this out automatically, we take the simpler route of requesting this information from the user. In the future we may add additional tests to the template to determine this settling time automatically. For some tests the test options data can be significantly more complicated. In the sample-and-hold template the testbench writer needs to know how to drive the clock. For some user circuits this is as simple as knowing the clock frequency, but other user circuits may

have multiple input clocks that are meant to follow a precise timing schedule. To address this the sample-and-hold template defines a format with which the user can pass the timing schedule in as a dictionary. This dictionary can also include limits on timing jitter between different clocks, which is treated as an optional input. With test option dictionaries, we encourage template writers to follow our guiding principle of providing reasonable defaults but advanced options - for example, the user should be allowed to enter only a frequency with no timing schedule if they only have one clock.

The first important limitation that the template writer needs to contend with is simulation speed. The short testbench section will be run many times, for some user circuits hundreds of times, with the optional inputs set to different values. This means template writers should keep the testbench section as short as possible without compromising the quality of the collected data.

The second limitation is the template writer's limited knowledge of what happens before and after the testbench. Fixture will compile many copies of the testbench together end-to-end before giving them to the simulator, and optional inputs are adjusted in the instant between these copies. The testbench writer should therefore keep in mind that just before the testbench starts could be the very beginning of the simulation, or it could be just after optional inputs (including true digital operating modes) were changed. The testbench writer is encouraged to assume that a significant change was just made before the testbench started and wait an amount of time appropriate to the circuit and the test before taking any measurements. For many tests this is not an issue, for example in the amplifier's DC test the testbench can apply the new input immediately because the testbench will wait for the output to settle before taking a measurement anyway. In other cases, such as the oscillator template, it is smart to wait a few cycles to allow the state of the circuit to settle before measuring. In general, modeling the timing of optional input effects is difficult, and is a limitation of Fixture. Section 5.3.3 discusses one strategy that template writers can use to measure the timing of optional effects in specific cases.

5.3.2 Vectoring a Testbench

Just as Fixture can automatically vector Parameter Equations (Section 4.5) it can also automatically vector the other sections of a template: the Testbench, Input Sample Space, and Analysis sections. The most challenging of these is the testbench, and this section will discuss some of the limitations when it comes to automated vectoring that arise because of the testbench section.

The easiest section to vector is the Input Sample Space. When a template writer specifies a particular input to be sampled it can be tied to a required input or output, in which case Fixture knows to vector those samples. To vector a sample, Fixture simply adds more dimensions to the sample space corresponding to the additional inputs and uses the same random constraints we have already discussed in Section 5.1.

The Analysis section of the test is also relatively easy to vector. This section turns raw measurements from the testbench into values that can be used in the parameter definitions (Section 4.1).

When the raw values are vectored, we can simply run the Analysis section multiple times, passing in the raw values for each component of the vector and collecting the processed values accordingly.

The most difficult section to automatically vector is the Testbench. When a user decides to vector a required input, Fixture takes advantage of python's dynamic typing to change the input object that gets passed into the testbench accordingly, as we saw in Section 5.2.2. To make this work, Fixture needs to change the objects that the template writer interacts with, specifically the required input pins and the sampled input values. From the user configuration, Fixture can tell which required input pins have been vectored, and can easily replace the template writer's references to those with references to the appropriate vector of physical pins. For the input values, Fixture can again easily replace single values with vectors from the Input Sample Space section. While these changes work nicely with the *fault* functions, the template writer may choose to interact with these vectored objects in other ways. Most commonly, the template writer may do some algebra with the sampled values before passing them to a `poke` function. In the sample-and-hold template, when the template writer wants to apply an input ramp with a particular slope they may add the sampled ramp start voltage to the sampled ramp step size to find the ramp end voltage. For simple algebra like this we can use the *numpy* library [50] to make basic algebraic operations apply element-wise to the vectors, without the template writer needing to adjust their syntax.

So far Fixture has been able to automatically vector everything we have discussed, but that is not always the case. Because the template writer is allowed to write arbitrary python in the testbench, there can always be situations where one can write a testbench that works correctly for a single input, but not a vectored input. For example, in Section 5.2.1 we discussed flow control based on sampled inputs, with the example of a sample-and-hold template that limits its ramp by voltage or by time depending on the magnitude of the sampled slope. Any flow control or variable delay that is a function of a sampled input will fail to be automatically vectored by Fixture because it is not clear how to allow the testbench to diverge for different components of a vector. If the testbench writer knows what should be done for their particular case they are encouraged to have the Testbench explicitly detect vectored inputs, outputs, or samples and handle those cases accordingly. We can consider the automatically-vectored testbenches to be the smart default, and the ability of a testbench writer to manually specify complicated cases the advanced option.

There are some automated vectoring strategies that might be able to handle a larger variety of testbenches than Fixture's current strategy. For example, Fixture could call the testbench function individually for each vector component, then apply each resulting set of stimuli to the circuit simultaneously. Unfortunately, this alternate method still fails when the testbench includes a timing delay that is different for different components of the vector. Because we believe there will always be cases that require explicit intervention from the template writer we did not spend a significant amount of effort teaching Fixture to automatically vector as many cases as possible. For now certain tests may be unavailable to user-vectored circuits, but hopefully in tests where vectoring makes sense the

library will eventually be improved to include a vectoring-compatible version.

5.3.3 Optional Input Timing

In most cases, we keep the details of optional inputs hidden from the template writer because the template should be written in such a way that it works for any set of optional inputs, including no optional inputs. There are some situations, however, where a particular test will only apply to user circuits with optional inputs. In these situations Fixture allows the test writer to request a particular optional input from the user, and then the test writer can treat it as they would treat a required input for that particular test.

We encountered such an situation when working with the DragonPHY project [51]. DragonPHY has a phase interpolator which can produce an output clock with any phase relative to an input clock. This block is implemented using a number of phase blenders which each handle a small slice of the 2π total output phase. Each individual phase blender has its output phase fine-tuned by a optional input signal. Because all the phase blenders together cover every possible output phase it is guaranteed that one can have its output edge at the same time as an optional input edge. This can lead to the glitching behavior shown in Figure 2.4. While creating models using Fixture we discovered that this glitch did occur in DragonPHY, and we were able to address the issue before tape-out.

To detect and model this glitching behavior in Fixture, we use a test in the phase blender template which sweeps the relative phase between the input clock edges and phase adjustment input edges while checking for a glitch. Of course, this test only applies to phase blender circuits which have a phase adjustment input, which is not present in every phase blender. The only way to have both a basic test for a phase blender with no adjustment input and a glitch test that measures timing of the adjustment input is to give the template writer access to optional inputs.

The way we accomplish this in Fixture is to rely on the test option dictionary described in Section 5.3.1. The user can use the dictionary to pass in the name of an optional input that corresponds to the adjustment input. With that name, the template writer can access the object corresponding to that optional input, and treat it the same as an additional required input. This gives the template writer the freedom to implement any *fault* testbench interacting with any of a circuit's inputs. The disadvantage is that the specific test accessing an optional input will be unavailable to user circuits without that input, so template writers should only use this functionality when it is necessary and allow Fixture's built-in optional input handling capabilities to model optional input effects as much as possible.

Chapter 6

Model Fitting

Once Fixture has decided on a model to use and collected simulation data for the user's circuit, it must fit model coefficients to the measured data. This can be challenging to get right because the model can be any function defined by the template writer or end user, and fitting the model coefficients becomes a nonlinear optimization problem. Fixture must also provide useful debugging information in cases where the model is not a good fit for the measured data. This chapter will explain the challenges with model fitting and Fixture's solution, as well as discuss the many plots and other outputs Fixture provides for debugging.

6.1 Regression

Fixture's ability to model arbitrary nonlinear circuit behaviors, both in the template and in the response to optional inputs, opens the door for the user to specify complex modeling equations. This complexity is good because it allows Fixture to handle a variety of user circuits, but it also introduces challenges for model fitting and debugging. This section will describe the approach Fixture takes to fit coefficients in arbitrary nonlinear equations and help the user debug when it cannot find a set of coefficients to make the specified equation match measured behavior.

6.1.1 A Challenging Example

To illustrate Fixture's approach to regression we will use a challenging example equation which contains nonlinearity in both the template equation and the user's requested dependence on optional inputs. A differential amplifier's output is modeled with a gain and saturating amplitude, and each of those two parameters is a function of `radj` and `vdd` optional inputs. This example is the same one used to illustrate dependence on optional inputs in Section 4.4.2, and we will reproduce Equation 4.8 here for reference:

$$\text{out} = \text{amplitude} \cdot f\left(\frac{\text{gain}}{\text{amplitude}} \cdot \text{in}\right) \tag{6.1a}$$

$$f(x) = \tanh(x) \tag{6.1b}$$

$$\text{amplitude} = \text{amplitude_nom} + \text{amplitude_vdd} + \text{amplitude_radj} \tag{6.1c}$$

$$\text{amplitude_nom} = A \tag{6.1d}$$

$$\text{amplitude_vdd} = B_1 \cdot \tilde{\text{vdd}} + B_2 \cdot \tilde{\text{vdd}}^2 \tag{6.1e}$$

$$\tilde{\text{vdd}} = \text{vdd} - \text{vdd_nom} \tag{6.1f}$$

$$\text{amplitude_radj} = g(\text{radj}) - g(\text{radj_nom}) \tag{6.1g}$$

$$g(\text{radj}) = \frac{1}{C_6 + \sum_{i=0}^{5} C_i \cdot \text{radj}[i]} \tag{6.1h}$$

$$\text{gain} = \text{gain_nom} + \text{gain_vdd} + \text{gain_radj} \tag{6.1i}$$

$$\text{gain_nom} = D \tag{6.1j}$$

$$\text{gain_vdd} = E_1 \cdot \tilde{\text{vdd}} + E_2 \cdot \tilde{\text{vdd}}^2 \tag{6.1k}$$

$$\tilde{\text{vdd}} = \text{vdd} - \text{vdd_nom} \tag{6.1l}$$

$$\text{gain_radj} = h(\text{radj}) - h(\text{radj_nom}) \tag{6.1m}$$

$$h(\text{radj}) = \frac{1}{F_6 + \sum_{i=0}^{5} F_i \cdot \text{radj}[i]} \tag{6.1n}$$

If we begin with Equation 6.1a and recursively apply all the definitions in 6.1b-n, we end up with one large equation that computes out as a function of the inputs `in`, `vdd`, `radj[5:0]`, and the coefficients A, $B_{1\text{-}2}$, $C_{0\text{-}8}$, D, $E_{1\text{-}2}$, and $F_{0\text{-}8}$. Fixture's goal is to use measured data for `in`, `vdd`, `radj[5:0]`, and `out` to find best-fit values for the coefficients that minimize the mean-square error between the measured and predicted `out`. Throughout the rest of Section 6.1 we will discuss why this problem is challenging and strategies for overcoming that challenge.

6.1.2 Challenges with Nonlinear Fitting

When the output is a linear function of all the coefficients the coefficients can be extracted using linear regression. In cases where the equation is nonlinear we must replace the linear regression with a nonlinear optimization routine. For the purposes of this discussion, a nonlinear optimizer is solving a minimization problem. It is given a function that computes an error as a function of coefficient values, and it searches for coefficient values to minimize the error. In Fixture, the coefficients are the circuit parameters, and the error is the mean square error (MSE) of the model predictions over

all the datapoints that were simulated.

In Fixture we used the *scipy.optimize* package [59] to minimize error using either the Broyden–Fletcher–Goldfarb–Shanno algorithm (BFGS) or the Nelder-Mead algorithm, and in both cases ran into the same three issues: the speed of the optimizer, the quality of results from the optimizer, and the ability to debug the results.

Issue: Optimizer Speed

Compared to the computational load of circuit simulation, we would expect the computation for model fitting to be small. The model is intended to be a replacement for a spice circuit, so by necessity it should be relatively easy to compute. The issue is that a nonlinear optimizer relies only on repeated evaluation of the function to tune the coefficients, and there can be a large number of coefficients to tune.

Let us consider an example circuit: a nonlinear amplifier with two 6-bit buses that affect its behavior. If the user does not specify which buses affect which behavior, Fixture will assume all can affect all, which is a reasonable first pass if a circuit designer wants to give the tool full freedom to find nonidealities that were not intended. The nonlinear Parameter Expression might have four parameters: `gain`, `out_max`, `out_min`, and `in_offset`. But each parameter has to capture a nominal value and the effect of 12 different input bits, for a total of 13 coefficients per parameter, or $N = 52$ parameters for the optimizer to tune. A high number of optimizer parameters has three issues. The first is that it takes a lot of input data points to reliably capture the effect of each parameter, so each computation of the error function is finding the MSE over many, typically at least $2N$, applications of the circuit model. The second is that many nonlinear optimizers rely on a numerical computation of the gradient at each step, and this takes on the order of N computations of the error function. Finally, a large N means that the optimizer is searching a high-dimensional space for the minimum, and we should expect more steps for it to converge to an answer. All these effects combine in such a way that the time it takes for the optimizer to converge grows quickly as a function of the number of circuit parameters and can quickly become problematic.

Issue: Optimizer Result Quality

It is well known that nonlinear optimization is a challenging problem, and the optimizers we are using are not guaranteed to converge to the global minimum. We found that in models with a large number of parameters, the optimizer rarely converged to an acceptable solution unless it was given an initial guess that was reasonably close to the expected solution. One reason this is so challenging for the optimizers is that the optimal values of the parameters often differ by orders of magnitude. For example, one parameter may represent an offset voltage on the order of millivolts, while another represents a current-to-voltage gain on the order of volts per microamp. Converted to standard units, these values differ by 10^9. Fixture could partially address this problem by using its

knowledge of the circuit type to guess common values as starting points for the optimizer. Rather than investigate this path, we base our initial guesses on the collected data directly, using a strategy described in Section 6.1.4.

Issue: Debugging Ability

Even if the nonlinear optimizer always ran instantly and always found the global minimum, there would still be an issue with this approach. In the case where the model does not fit the circuit behavior no matter the coefficients, for example because the user specified a linear relationship for an effect that is really nonlinear, the debugging is very difficult. The issue is most apparent when the output is affected by two optional inputs, where one is properly modeled and one is improperly modeled. In these cases it can be difficult to tell that the properly-modeled input is correct because of the error contributed by the incorrect portions of the model. In addition, when there is little data to fit to the optimizer may move the correctly-modeled portion away from its optimal coefficients in order to make up for shortcomings from the incorrectly-modeled portion. In the next section we will present an example of an incorrect model and the challenges associated with debugging it.

6.1.3 An Unsuitable Sampling Approach

When choosing which points to simulate in order to collect data to fit model coefficients, there is a tradeoff between minimizing simulation time and maximizing information to help with fitting. Before Fixture, the existing DaVE tools focused on minimizing simulation time. To fit a model with N coefficients, DaVE simulated on the order of N points according to the random constraints described in Section 5.1. In this section, we will attempt to debug a bad fit with this old approach to demonstrate why a better approach is necessary.

For demonstration, we will attempt to model our motivating circuit's output using a linear model. Recall that the gain of the motivating circuit is inversely proportional to the value of the `radj` bus. The model we will attempt to fit for the gain and offset is simply a linear function of `vdd` and each bit of the `radj` bus:

$$\begin{aligned} \text{out_diff} = \quad & \left(A + B \cdot \text{vdd} + \sum_{i=0}^{7} C_i \cdot \text{radj}[i]\right) \cdot \text{in_diff} \\ & + \left(D + E \cdot \text{vdd} + \sum_{i=0}^{7} F_i \cdot \text{radj}[i]\right) \end{aligned} \tag{6.2}$$

Note that for clarity we omit the differential output's dependence on the common-mode input, which is typically small. The plots in this section were produced with the full equation, including that effect.

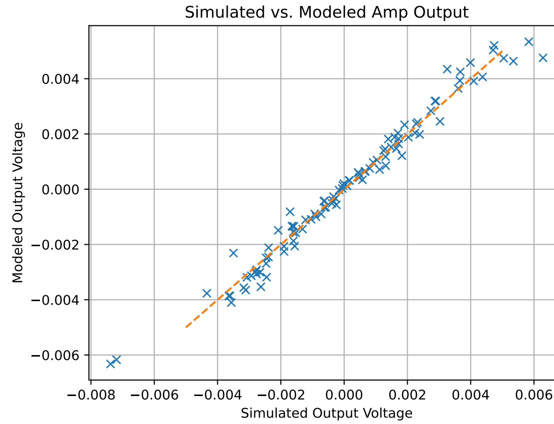


Figure 6.1: The results of applying the model in Equation 6.2. If the model were a perfect fit then all the points would lie along the dotted line where modeled output equals simulated output. Instead, we see significant deviation from the dotted line. This is what we expect, since we used a linear model but we know the resistor behavior is nonlinear.

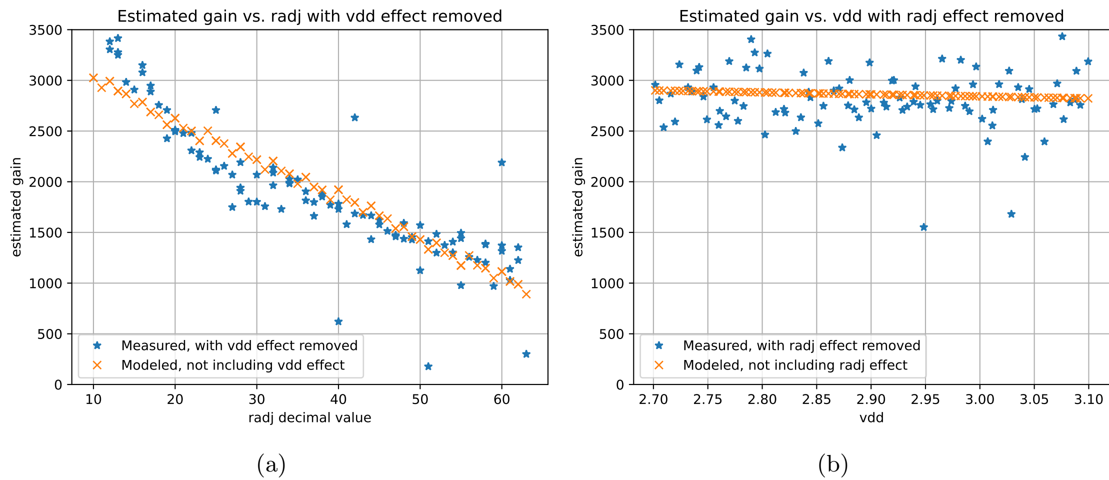
Because Equation 6.2 is linear with respect to each coefficient we can find the optimal fit using linear regression. Doing this, we find the 20 best-fit numeric values for the coefficients A through F_7. We visualize how well this model matches the circuit's behavior in Figure 6.1, and we see that it is not perfect because it cannot capture the nonlinear behavior of the feedback resistor.

If the user does not know why the model fits poorly, what steps could they take to debug this model? One reasonable approach is to attempt to visualize the effect of individual optional inputs on individual parameters. Using only the data from Figure 6.1 and the best-fit coefficients from Equation 6.2, we can estimate these individual effects. The idea is to rearrange the equation to put the desired quantity on the left hand side. For example, if the user is interested in the effect of the `radj` input on the gain, we can rearrange the equation as follows:

$$A + \sum_{i=0}^{7} C_i \cdot \text{radj}[i] = \frac{\text{out_diff} - (D + E \cdot \text{vdd} + \sum_{i=0}^{7} F_i \cdot \text{radj}[i])}{\text{in_diff}} - B \cdot \text{vdd} \tag{6.3}$$

The left hand side of Equation 6.3 represents the nominal gain plus the effect of the `radj` bus. We can check our fit by evaluating the left hand side of the equation with the best-fit values of A and C_i to see what the model predicts for the gain and compare it to the right hand side using our measured inputs, outputs, and our best-fit parameters. With a good fit, parameters E and B would remove the effect of `vdd` variation from the right hand side. Performing this process for both `radj` and `vdd`, we can come up with the plots in Figure 6.2

Considering the measured points in Figure 6.2a the reciprocal relationship between `radj` and gain is visible, but there is also a lot of noise in the plot. While the user may correctly deduce from



(a) (b)

Figure 6.2: Linear model of the motivating circuit, showing both the measured data with the best estimate of the gain with other optional effects removed, and also the corresponding linear model predictions for gain. Note that the removal of vdd effects in (a) and removal of radj effects in (b) are problematic because they depend on an imperfect model, leading to "noise" in the measured datapoints that is not a property of the real circuit. For that reason, Fixture does not produce these plots and instead replaces them with the plots in Figure 6.6.

this plot that the linear dependence on `radj` is not sufficient, they may also be confused by the noise and outliers. Considering Figure 6.2b, the user may assume that the model for gain with respect to `vdd` is bad, or that there is an additional unmodeled effect causing the gain to vary significantly over multiple datapoints. In reality, the only issue with the model is gain's dependence on `radj`, and shows up in the gain vs. vdd plot because of the way we used the bad model to try to remove the effects of **radj** from the measured data. In addition, the way that gain is calculated in Equation 6.3 involves division by `in`, and for a differential amplifier `in` is allowed to be nearly or exactly zero. This effectively amplifies small errors in the model, leading to the outliers in Figure 6.2.

Our conclusion is that this method of removing the effects of individual optional inputs from data where all optional inputs were varied simultaneously is only effective if the fit is good, and that is a situation where this breakdown isn't needed. For situations where the fit is poor, this approach doesn't help isolate the problem. When the fit is poor, in order to properly separate the effects of different optional inputs we would need to already know what the effect of each input is. Fixture sidesteps this issue by collecting data in a way that keeps these effects separate from the start. Fixture's approach, including the plots it produces, will be described in the next section.

6.1.4 Fixture's Improved Approach

To overcome the difficulty of fitting and debugging arbitrary nonlinear models, we turned to our guiding principle and considered how engineers typically fit models to observed circuit behavior. The key is to consider the effects of inputs one at a time, and then combine these individual effects for the final model. To do so we will need to collect datapoints differently, testing specific combinations of inputs that let us find the coefficients individually or in small groups. This helps the nonlinear optimizer because it can fit subsets of the coefficients without considering the entire modeling equation at once. This change also helps with debugging since issues with modeling one input no longer affect the modeling related to another input. It also helps with plotting since plots typically only have one independent axis, and with this change we simulate circuit effects with respect to one changing input at a time.

There are two keys to Fixture's approach. The first is to sweep one optional input at a time while holding the others at their nominal value. The second is, during the sweep of an optional input `opt`, to hold `opt` at a particular value temporarily while sweeping the required inputs. The number of sample points needed in each sweep depends on the number of coefficients being fit; empirically, 3 times the number of coefficients that need to be fit with the data works well. Each time we choose a set of random values we are doing so according to the constraints described in Section 5.1.

To illustrate this strategy we will describe the sampling for the challenging example from Section 6.1.1. We will refer to the variables $N_{1\text{-}4}$ for the number of points in various sweeps, and their meanings are summarized in the table below.[1] Finally, Figure 6.3 shows a shortened list of sample points chosen according to this strategy, and the parameters extracted from each sweep.

Name	Value	Swept Input	Coefficients to fit
N_1	6	`in`	gain and amplitude
N_2	9	`vdd`	a parameter's nominal value and its dependence on vdd and vdd^2
N_3	24	`radj`	a parameter's nominal value, plus 6 bit weights and an offset for the nonlinear function of the bus
N_4	60	all inputs	all 20 coefficients in Equation 6.1

At a high level, Fixture's sampling is broken into two phases: the optional input sweeps and the simultaneous sweep. In the first phase we do an individual sweep for each optional input. During the sweep for a particular optional input we move that optional input as well as all the required inputs. Then for the second phase we move all optional and required inputs for each sample.

For our example circuit, we begin by sweeping `vdd` while holding `radj` at its nominal value. We choose N_2 random values for `vdd`, and for each one of those values choose N_1 random values for

[1] Equation 6.1 is simplified because it leaves out dependence on the common mode input. When characterizing the motivating circuit, Fixture vectors `in` according to the rules in Section 4.5, adding additional coefficients and changing the values depending on `in` to N_1=9 and N_4=90.

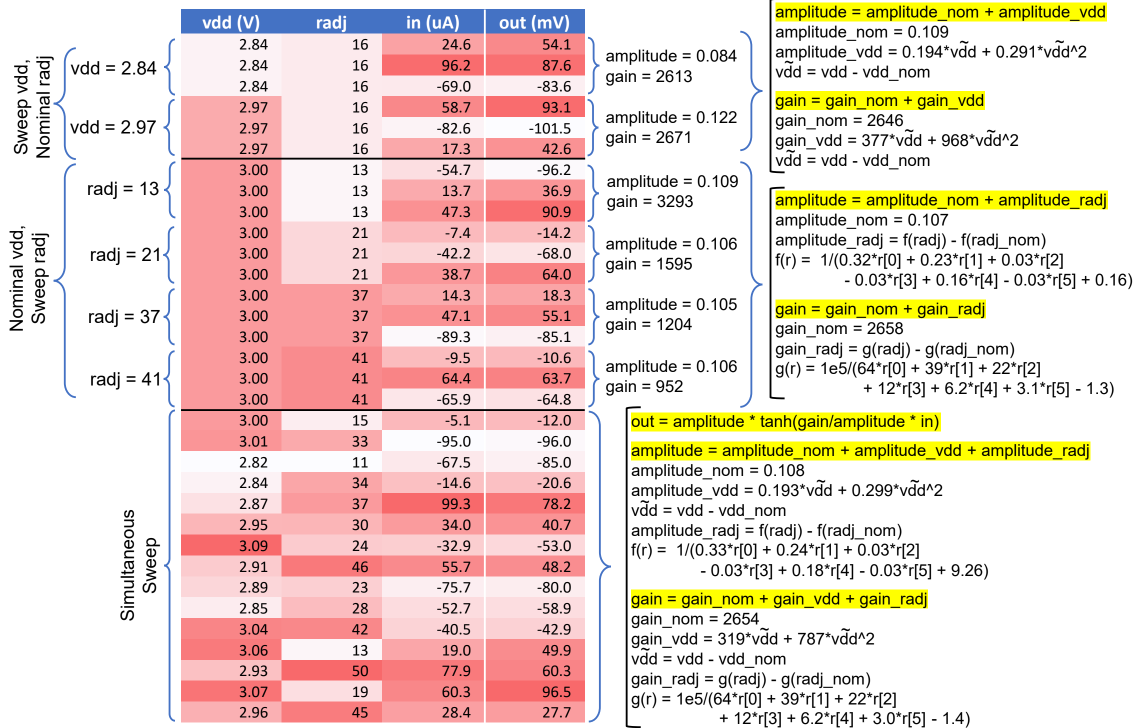


Figure 6.3: Example data collected for a differential amplifier circuit with two optional inputs, `vdd` and `radj`, to illustrate the way Fixture chooses input sample points. In this figure the total number of sample points has been reduced for easier illustration, and the best-fit values were extracted from a simulation using a larger number of points. In the "Sweep vdd" section 2 values have been chosen for `vdd`, and for each value there is a 3-datapoint sweep of `in`. This allows Fixture to extract the gain and amplitude separately for each value of `vdd` and then combine them to find the amplitude and gain in terms of `vdd`. In the simultaneous sweep section, Fixture fits one large equation including all the coefficients, rather than fitting in pieces. The results from the individual sweep sections are used for plots, debugging, and finding an initial guess for parameter values; only the results from the simultaneous sweep are used in the functional model.

`in`. Considering just one set of N_1 points where `in` has been swept but both optional inputs are frozen, we can apply our nonlinear optimizer with Equation 6.1a to estimate values for the gain and amplitude. We repeat this process N_2 times to get N_2 different estimates for amplitude and gain, each corresponding to a different value of `vdd`. Finally, we can take these N_2 estimates of the amplitude and apply the nonlinear optimizer with Equations 6.1c-h. Because `radj` is at its nominal value, we know 6.1g is zero and can drop the corresponding terms from the equation. This reduces the number of coefficients the optimizer needs to fit to just three: A, B_1, and B_2. To complete the sweep of `vdd` we repeat the fitting process for the gain to estimate D, E_1, and E_2. In total, we have measured $N_1 \cdot N_2$ points so far.

Next, this entire process is repeated for `radj`. This produces estimates for A, $C_{0\text{-}6}$, D, and $F_{0\text{-}6}$. Notice that we have two estimates for A and two for D. This will be resolved in the final step, when we use the simultaneous sweep. In this step we have measured an additional $N_1 \cdot N_3$ points, for a total of $N_1 \cdot N_2 + N_1 \cdot N_3$ so far.

In all the datapoints we have sampled up to this point we have swept only one optional input at a time, while leaving the other at its nominal value. In general this does not fully exercise the circuit. For example, the maximum output voltage may only be achieved when both amplitude and gain are at their highest values, which may occur only at the extreme values of `vdd` and `radj`. In many circuits nonlinear effects are most significant when voltages are at their most extreme values, so it is important to sweep all inputs simultaneously. In our last phase we do exactly that, choosing N_4 datapoints where all optional and required inputs are swept simultaneously according to the constraints in Section 5.1. We must use the nonlinear optimizer to fit these points using the entire set of equations together. The way to get a good result from the optimizer is to give it an initial guess that is close to the maximum value; we can use our individual fits from the previous phases to get this initial guess. For coefficients A and D we take the average of their two estimates. In total, we take $N_1 \cdot N_2 + N_1 \cdot N_3 + N_4$ sample points throughout this whole process.

In summary, we split the task of fitting a nonlinear equation into smaller pieces. This follows our guiding principle of approaching the problem the way engineers already do. Analog designers consider the effects of optional inputs one at a time, so we have Fixture do the same thing. Only once we have some understanding of how the inputs affect circuit behavior individually do we try exercising them together, and we use our results from individual fits to help with our combined fit.

6.1.5 Additional Nonlinear Fitting Techniques

The approach described in the previous section works well for fitting the simultaneous sweep, but in practice the nonlinear optimizer sometimes fails to converge for the small individual fits. We have found several strategies to help with this issue.

Basin Hopping

The first improvement is to use a nonlinear optimizer with a basin hopping technique. In short, when the optimizer finds a local minimum it will try perturbing the inputs randomly to hopefully "hop" into a different location with a different local minimum. This improvement was the easiest to implement because we were able to use an existing basin hopping implementation in the *scipy* library. The disadvantage is that this technique takes longer to run than a traditional optimizer because it is running repeatedly in different basins.

Sharing Fitting Results

Considering the example sampling pattern in Section 6.1.4, we make B calls to the nonlinear optimizer to find amplitude and gain as we sweep `vdd`. Although those values change as `vdd` changes, the variation is small compared to the total space the optimizer could be searching. For this reason, Fixture uses the result of the previous fit as the starting point for the next fit, and even goes through the list of B fits twice so that improvements in the solution the first time around can be propagated to all the fits in the second time around. This method works well along with basin hopping, because it essentially gives the optimizer many more opportunities to randomly hop into a good basin, and then share that good result with the other fits.

Fit Tricks

For some equations we know a specific fitting technique that works much better than the nonlinear optimizer. The most important example of this is an equation that is linear with respect to each of its coefficients. In that case we can use linear regression to find the optimal coefficient values very efficiently. How does Fixture know when it can apply linear regression? We created a library of "Fit Tricks," each of which includes a method to recognize algebraic expressions it can fit and a method to perform fitting. The entries in the library inspect the Abstract Syntax Tree (AST) of the equation being fit and apply their technique if the AST matches a specific form.

Init Tricks

The "Init Tricks" are very similar to the Fit Tricks except that they only give an estimate of the final fit. This is then used as a starting point for the nonlinear optimizer.

One example of an Init Trick is the reciprocal trick. It matches expressions of the form:

$$\text{measured_data} = \frac{1}{f_{\text{linear}}(c_0, ...)} + g(c_1, ...) \tag{6.4}$$

It applies the approximation:

$$\frac{1}{\text{measured_data}} \approx f_{\text{linear}}(c_0, ...) \tag{6.5}$$

It then uses linear regression to estimate the coefficients of the linear function. These are not perfect since it has neglected the effects of $g(c_1, ...)$, but it serves as a good starting point for the nonlinear optimizer. This technique works extremely well for the effect of `radj` on gain in our example, Equation 6.1m.

Accepting Help

In cases where none of the above methods work, Fixture can rely on the user to provide an initial guess. We try to avoid this method because it is extra work for the user, but it should not lead to any errors because Fixture is only using it as an initial guess and will still fit to the actual data.

6.2 Plotting

Each time Fixture is run it produces many plots automatically using the *Matplotlib* library [60]. Fixture's approach is to ensure that no matter what question the user has about the model there will be a plot already produced to answer that question. In this section we will walk through the different plots that are produced and discuss why they can be more helpful than text-based results alone.

6.2.1 Fixed Optional Input Plots

In Section 6.1.4 we described a particular order in which Fixture chooses sample points and runs regression on a subset of the modeling equations. Fixture produces plots of these intermediate fitting results to help the user understand circuit behavior and debug any issues. Figure 6.4 shows four of these plots for a well-fitting, nonlinear model.

These individual plots, with the optional inputs frozen for each sweep of the inputs, are the first step Fixture takes when looking for optional model coefficients. If the final model is not as accurate as the user had hoped, it is often useful to come back to this first step to see whether the error is with the circuit itself or with the model.

In all the plots in this section we have used the differential input as the independent axis. Although Fixture does sweep the optional inputs one at a time, it always sweeps the template inputs - in this case differential input and common mode input - together. So all the measured and modeled data in this section is with respect to both differential and common mode data, but the common mode is not visible. In Fixture, plots are produced with respect to all axes, so the same plots are available with respect to common mode input as well, but they are not as useful because

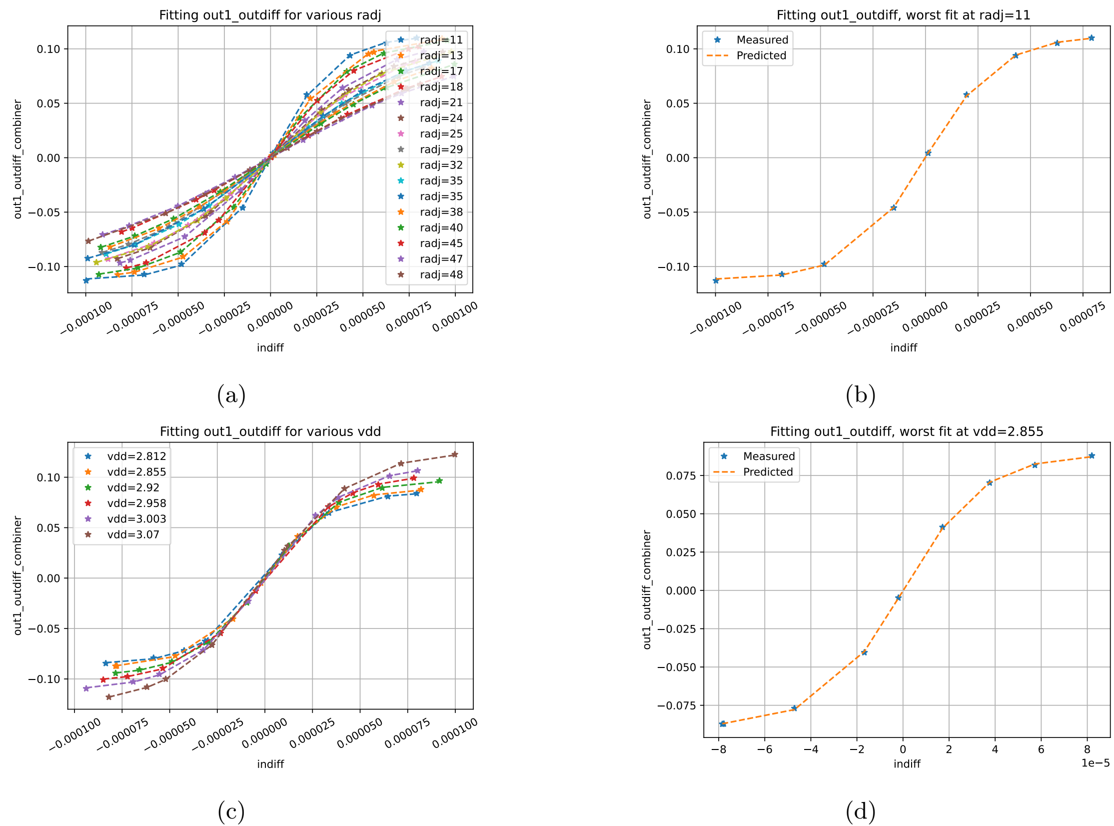
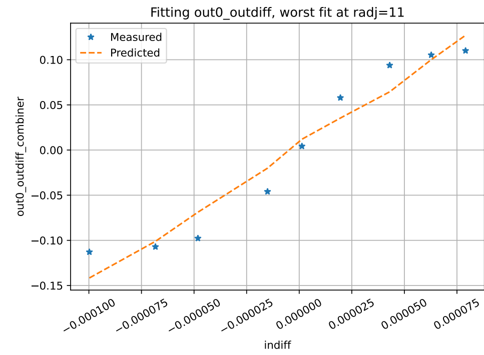
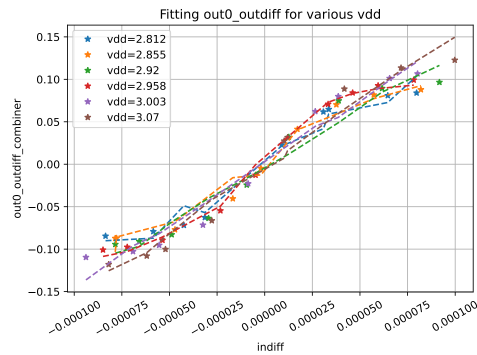


Figure 6.4: (a) Many individual fits of differential output vs. differential input, each with a different value of the `radj` optional input. Immediately the user can see the range of gains in the linear region near the origin as well as the saturating behavior at the extremes. Additionally, because the dotted lines (model predictions) match well with the stars (measured data), the user knows that the model from input to output is good. (b) Fixture automatically selects the case from plot a with the worst mean-square error and plots it individually so the user can see it clearly. In this case the fit is good. (c) Similar to the first plot, but for sweeping `vdd`. We can see that the gain is not affected significantly by `vdd`, but the amplitude is. Note that there are fewer curves in this plot than in the `radj` version because there are fewer coefficients to fit with respect to `vdd` in the next step. (d) We see that the worst fit with respect to `vdd` is still accurate.

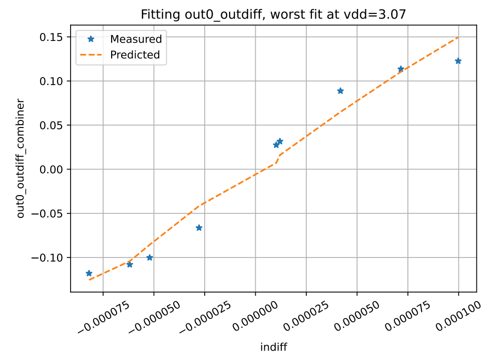
(a)



(b)

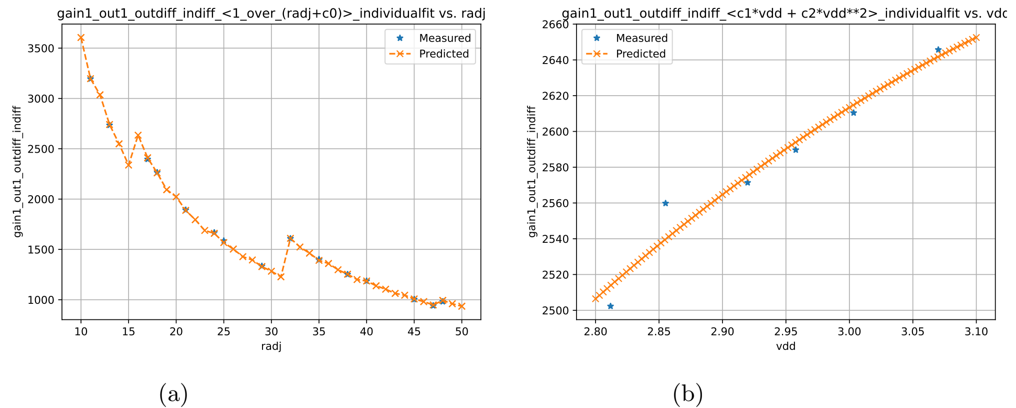


(c)



(d)

Figure 6.5: These figures show the same data as Figures 6.4a-d, except that the nonlinear model (Equation 6.1a) has been replaced by a linear one. The poor fits are apparent in figures (a) and (c), but the large number of overlapping plots makes them difficult to debug. The utility of plots (b) and (d) is clear, as they allow the user to see the linear shape of the prediction against the saturating, nonlinear shape of the measured data. Note that the predicted output does not appear perfectly linear in these plots because it is also a function of the common-mode input, which varies between test points.



(a) (b)

Figure 6.6: (a) gain vs. radj. This relationship is the motivation for the reciprocal relationship for radj. The fit is essentially perfect, modeling both the curved relationship and the sharp steps due to poor switch sizing in the design of the feedback resistor. (b) gain vs. vdd. We see that the predicted curve broadly matches the trend of the data, but the fit is not perfect. In this case it is not clear what the source of the error is, however, considering the scale of the vertical axis on both of these plots the error is very small compared to the range of possible gain values.

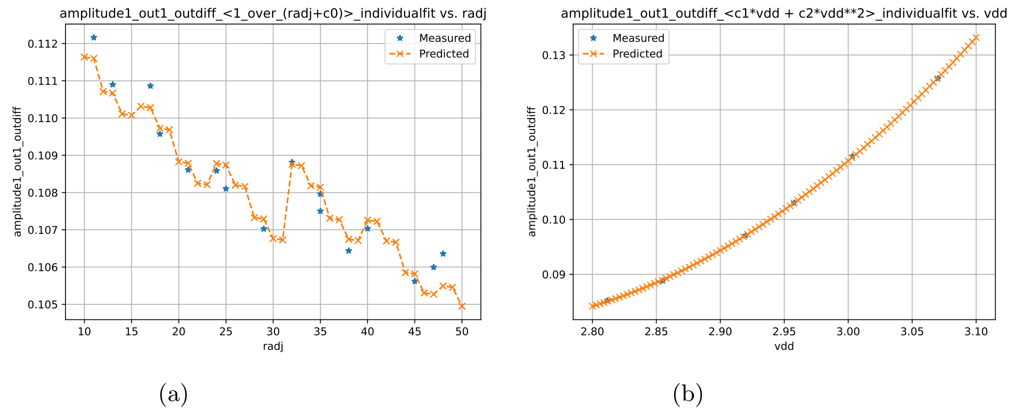
the effect of differential input on differential output is much stronger. We will see in Section 6.2.4 how Fixture uses contour plots to visualize two independent axes at once.

The hidden common mode input dependence is apparent in Figures 6.5b,d. Although the model is linear, the predicted data does not follow a perfectly straight line. This is because the model includes some erroneous effect of common mode input on differential output. The effect appears because the model cannot accurately express the output with the differential input alone, and there are few enough datapoints that the model begins to overfit using common mode input as well. This effect can be reduced by simulating more datapoints; however, in this specific instance it is already clear that the user should pick a different model if they are interested in a more accurate fit.

6.2.2 Parameter vs. Optional Input

The next step in Fixture's fitting of model coefficients is to fit each individual parameter with respect to each optional effect. Fixture will produce a debugging plot for each of these fits. Each datapoint in these plots is a result from one fit from the previous section. See Figure 6.6 for plots of gain vs. optional inputs.

Because Fixture does not have an estimate of what errors are important to the user, it always scales its plots to show as much data as possible. This means that axes for two different datasets may be scaled differently even though they represent the same quantity. Indeed, in Figure 6.6 the plot with respect to `vdd` is scaled up significantly, so errors that are likely insignificant seem large. Still, for users interested in the highest possible accuracy this error may be significant, so it is important



(a) (b)

Figure 6.7: (a) amplitude vs. radj. The fit is not perfect; looking at the vertical axis the variation of amplitude is small but probably not negligible. It is up to the user to decide whether this reciprocal model is good enough for their purposes, or whether they should use a different optional input dependence for this relationship. (b) amplitude vs. vdd. The quadratic model fits the data nicely.

that Fixture shows it.

If the user is interested in tracking down the source of an error in a parameter sweep plot like Figure 6.6, one place to start is the plots shown in the previous section, Section 6.2.1. In those plots, the user can see the source of each of the datapoints in the parameter sweep plots. For example, the measured point in Figure 6.6b that is higher than the predicted curve corresponds to the vdd $= 2.968$ curve in Figure 6.4c, and since that curve is not exceptional we can say the point is not an outlier because it is the result of a bad fit. In this case, the errors are small enough that they are likely at the limit of simulator accuracy. If the user wants to investigate further they can refer to the raw data, which will be discussed in Section 6.4.

In Figure 6.7a we have a fit that at first glance appears imperfect, and in Figure 6.7b a fit that is nearly perfect. Comparing the axes between the two plots, we see that the scale of the y-axis in the first plot is much smaller. In fact, the entire range of amplitude values as `radj` is swept is less than 10% of the amplitude, and the largest error is well under 1%. This is an example of a plot that is likely much more useful than its corresponding mean-square error value alone. While the mean-square error would tell the user that the fit is good, it would not indicate that the total effect of `radj` is small, which is important because the user may decide that a linear or even constant model for amplitude vs. `radj` is sufficient. In Figure 6.7b, seeing the figure rather than just the mean-square error gives the user an intuitive sense of how much the quadratic term helped the fit, and how much the error would suffer if the quadratic term were removed.

Finally, in Figure 6.8 we see an example of a nonlinear optimizer debug plot. For this specific effect, Fixture used the Reciprocal Init Trick (Section 6.1.5) to find the initial point to start the nonlinear optimization. To help with debugging, Fixture plots the estimated fit after the Init Trick

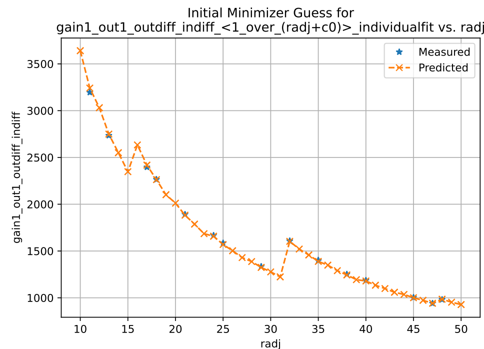


Figure 6.8: Result of using the Reciprocal Init Trick on the gain vs. radj data. This initial fit was used as the starting point for a nonlinear optimization that produced Figure 6.6a. That figure is nearly identical to this one because the Init Trick found a good fit and there was not much left for the optimizer to do.

but before the nonlinear optimization. In this case the fit was already nearly perfect after the Init Trick. This is because, in the language of Equation 6.4, the $g(c_1, ...)$ term is nearly zero for the `gain` vs. `radj` relationship.

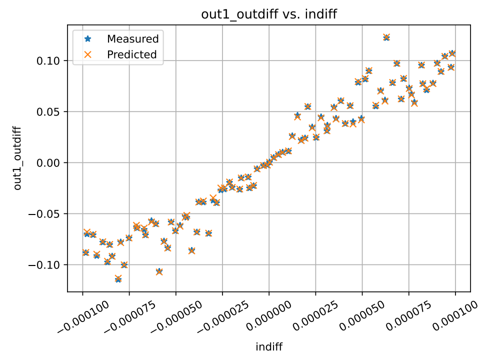
6.2.3 Final Model

After Fixture finishes estimating each of the model coefficients one optional input at a time, it uses those estimates as the starting point for the final nonlinear optimization with all the inputs swept simultaneously. Fixture then uses the final model to produce several more plots so that the user can determine whether the final model is appropriate for their use case. In these plots, the most useful visualization is to see the residual error between the modeled and measured output plotted against various inputs. If the model is poor with respect to one input, that should show up as a trend in the corresponding plot. Figure 6.9 shows the measured and predicted values as well as their residual error against the template inputs. Figure 6.10 shows the final model against optional inputs.

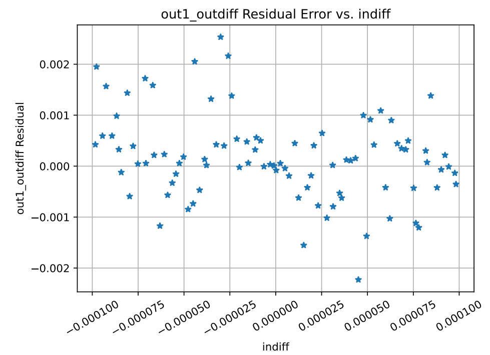
6.2.4 Contour Plots

Fixture uses contour plots to display data with two independent axes simultaneously. It can take some practice to read contour plots effectively, but in many cases they are worth studying because they contain more information than the scatter plots we have seen so far. There are a few rules that apply to all the contour plots produced by Fixture:

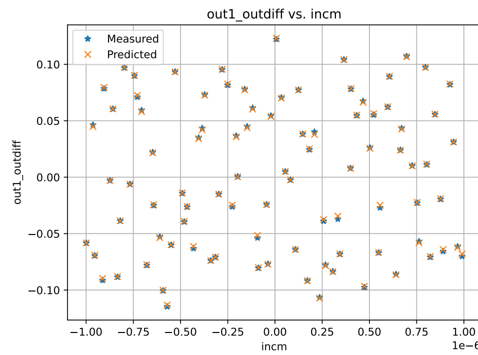
- Solid lines and colors correspond to measured data.
- If the plot has predicted data, predictions will be displayed with dashed lines only. If the predictions are perfect then the dashed lines cannot be seen because they fall directly on top



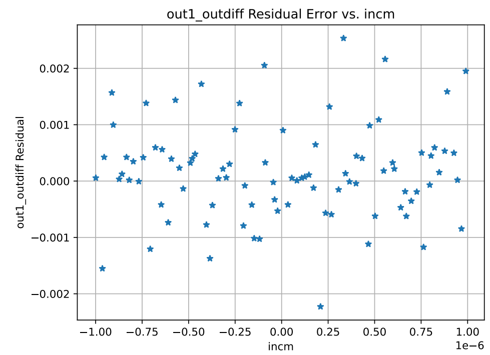
(a)



(b)

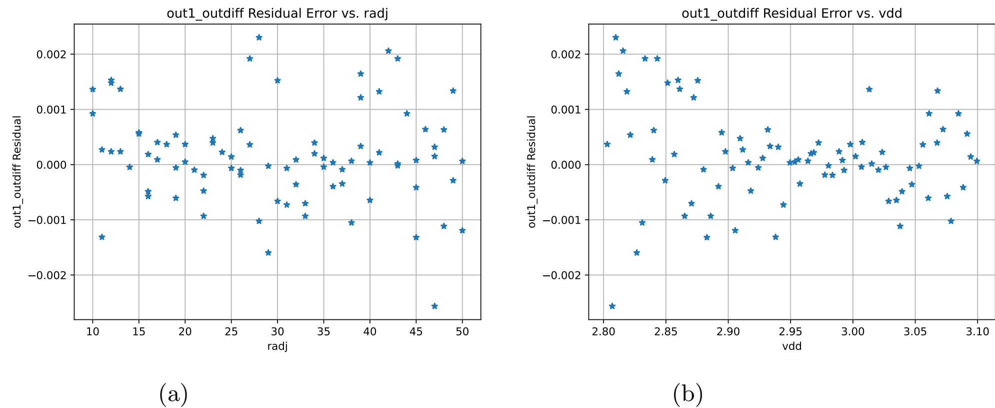


(c)



(d)

Figure 6.9: (a) differential output vs. differential input. Two datapoints collected with the same differential input can have a different differential output because the optional inputs are also being varied randomly. The close matching of the measured and predicted points means that this is a good model even when all the inputs are varied simultaneously. (b) Residual error for the differential output vs. differential input plot. This makes it easier to see the magnitude of the error and see whether there is any trend to indicate a systematic error. (c) differential output vs. common mode input. This plot is the least useful of the four since the common mode input does not affect the differential output very much. (d) Residual error for differential output vs. common mode input. This plot is useful for checking whether there is some small unmodeled effect of common mode input. In this case there is no significant trend, so the small errors are probably not due to unmodeled common mode effects.



(a) (b)

Figure 6.10: (a) differential output residual error vs. radj. Like Figure 6.9, the datapoints were collected while varying all the input simultaneously, and in this plot they are organized by the radj input to check for a systematic source of error. (b) differential output residual error vs. vdd. In this plot we do see a vague trend with error being worse further away from the nominal vdd value of 3.0. The maximum relative error of this model is less than 3%, but if the user is interested in a more accurate model this figure suggests looking at vdd effects first.

of the solid lines from the measured data.

- Blue "X"s correspond to datapoints. The smooth contours are made by interpolating between datapoints, but sometimes it is helpful to know where the measurements exist and where the displayed value is only an interpolation.

Figure 6.11 shows a few contour plots with the same example circuit we have been using. Because we have a good fit for this example circuit the residual error plots are not interesting. Let us consider the same circuit, but we will adjust the input range so that the saturation behavior is only significant when both radj and indiff are at their extreme values. Then, if we attempt to use a linear model, we might expect to see error appear only in specific corners of the contour plot. Looking at Figure 6.12, we actually see error all along the edge corresponding to high gain because the coefficient fit has sacrificed accuracy for small differential input in order to not overshoot too much for large differential input. Contour plots have a significant advantage over scatter plots when trying to visualize data with two important independent axes, such as the differential input and `radj` optional input in this example.

6.3 Additional Outputs from Fixture

In addition to plots, Fixture also gives a human-readable version of the model and several other debugging outputs. The user can also request the model in a format that can be read by mGenero to produce a functional model.

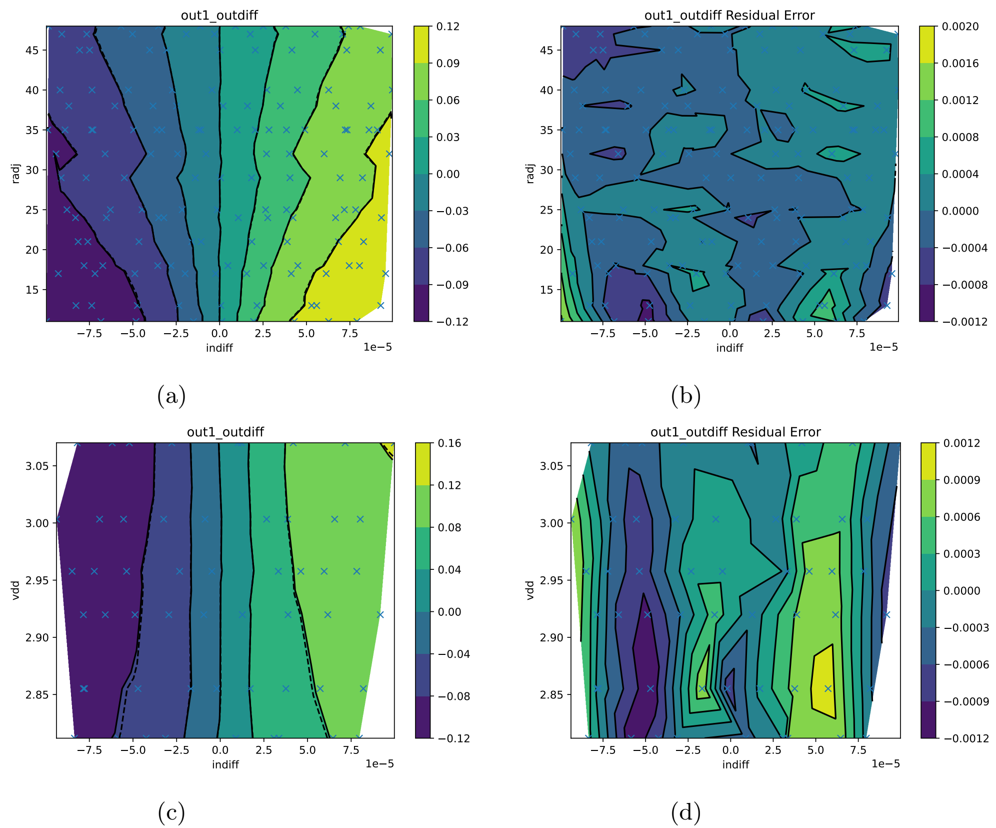


Figure 6.11: Contour plots showing the differential output or the residual error from modeling the differential output against various inputs. In this case the model fit is good, and the plots do not show any systematic trend in the error. For (a) and (b), the independent axes are the differential input and radj, so Fixture uses the data from the sweep of `radj`. (c) and (d) use the data from the sweep of `vdd`. Notice that the location of the measurements ("X"'s) fall into six horizontal lines; these correspond to the six values in the sweep of `vdd` seen in Figure 6.4c.

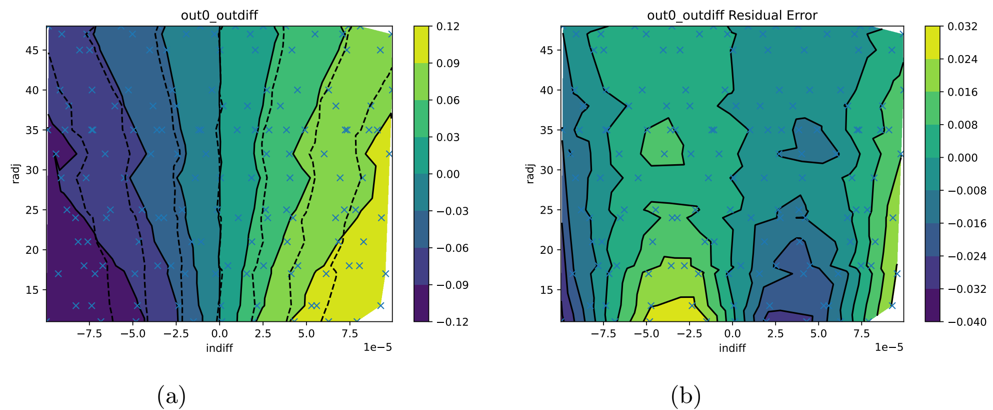


Figure 6.12: Contour plots for a non-saturating model of a saturating amplifier. The error in the model is difficult to visualize with only one independent axis, but is clear in these contour plots if the user takes the time to understand them. (a) The dotted lines represent the differential out model. At the bottom of the plot low `radj` causes high gain which causes saturation, but the model is not able to capture that saturation and diverges from the measured values. (b) The error very clearly has larger magnitude at low values of `radj`.

Input Samples

As explained in Section 6.1.4, Fixture's method of choosing input points is fairly complicated. Once it has chosen all the input points over all sweeps, it dumps those points in a csv file so that the user can inspect and debug with them. For ease of interpretation and in case the user wants to create plots in an external program, Fixture reports columns for each individual bit of a bus as well as the decimal interpretation of that bus according to the format the user specified in the user configuration file. In addition to a column for each input, the file also has columns to specify which optional input is being swept at that point (if any) and the index of the point in the sweep.

Testbench Input and Output

After fault compiles the testbench to the appropriate format, it saves that to file and calls the simulator to operate on that file. This means that both the file for the testbench in spice or Verilog and the raw testbench results are available in the filesystem for the user to inspect.

Extracted Data

After the simulation finishes, Fixture extracts data from the testbench results using the Template's analysis methods. If there is a bug with the way Fixture is interpreting testbench results, this would be one way the user could see the issue before it propagates to the final model. Additionally, if the user wants to use an external program to make any additional plots not already provided by Fixture they can use this file to do so.

The Final Model

Once Fixture has found the best-fit coefficients it will report the final model to the user. Because the model can be relatively large for a circuit with many optional inputs, Fixture lists it in a hierarchical format like Equation 6.1. The best-fit values for each coefficient are listed in a separate table. In this format, it is easy for the user answer any specific questions they might have about the model. Even if they are not familiar with the names of parameters used by Fixture they can easily see the definitions by looking at the top equations in the hierarchy.

In addition, if the final model is compatible with the mGenero library the user can request that the model be exported in an mGenero-compatible format for automatic model generation. If there is any translation necessary between the Fixture template's name for a parameter and mGenero's name for an equivalent parameter, that translation can also be supplied in an mGenero configuration file alongside the user configuration file.

6.4 Fixture Checkpoints

In addition to their use for debugging, Fixture can also use the intermediate outputs described in Section 6.3 to restart from the middle of its operation. Specifically, we split the operation of Fixture into four phases: `choose_inputs`, `run_sim`, `run_analysis`, and `run_regression`.

The checkpoint feature is primarily useful if the user makes a change to the user configuration file, but does not want to re-run the simulation if it is not necessary. This commonly happens because the user wants to change one of the optional input dependencies to try modeling their circuit in a different way.

The user can also use the checkpoint feature to customize the way Fixture runs. For example, if the user knows there is a particularly problematic combination of optional inputs they can manually add sample points to the table of input samples to exercise that problem area of the circuit. All the user has to do is add rows to the table and Fixture will take care of adjusting the testbench, analyzing results, and plotting models automatically.

6.5 Preliminary Model Generation

One final way the user can use the checkpoint system is to generate a preliminary model for a circuit before the spice model has been developed. During the development of an SoC, it is common to have analog and digital teams working on their portions of the SoC simultaneously. When the digital team is developing hardware that interacts with analog blocks it is extremely helpful for them to have a digital model to test with, even if the analog implementation is still a work in progress.

Within the DaVE ecosystem it is possible to use a templatized Verilog funcitonal model to generate a preliminary model based on a hand-written set of parameters. In many cases, however,

the specification that analog and digital teams work from is not a list of parameter, such as gain's sensitivity to the bits of an adjustment input, but rather a table of adjustment input values and corresponding gain values. With Fixture, we can insert this hand-written spec table in place of extracted testbench results, and run model fitting, plotting, and model generation as normal. This way, engineers get a visualization of how the few datapoints in the table were extended to the entire input range, as well as the Verilog functional model that digital engineers can use to develop their hardware.

There are some additional challenges that arise because a spec table typically lists parameter values rather than inputs and outputs, and because it usually has a relatively small amount of data to extrapolate from. The first issue is easily solved by adding new parameter definition equations to the template. It is straightforward to add an additional Test to an existing template with these equations since the other sections of the Test can be left blank.

Next, we have the problem of limited input data. Typically, this is only an issue when the number of coefficients in the model is close to or greater than the number of sample points in the spec table. The time this is most likely to happen is with digital adjustment buses. Fixture typically assigns one coefficient to each bit in a bus, which can easily be more than the number of example [bit vector, parameter value] pairs in the spec table. We solve this issue by introducing new styles of digital buses called `forced_binary` and `forced_thermometer`. Like the digital bus styles described in Section 6, this is a datatype that the user can assign to an input bus in the user configuration file. The difference is that during model fitting, the default dependency on this input will not be linear with respect to each bit, but rather linear with respect to the digital value of the bus. Instead of adding N coefficients for each N-bit bus, this strategy adds just 1 coefficient per bus.

For teams that already use Verilog functional modeling as part of their existing workflow, and are wary about switching to the DaVE environment without evidence that the models are accurate enough for their verification needs, preliminary models can be a good place to start. Having an inaccurate preliminary model is relatively low-impact because the mistake will be caught when the team moves to the final Verilog functional model. Still, using a preliminary model makes for a good trial run of models from the DaVE environment since the team will likely work with them for an extended period of time, and can easily compare their performance to the hand-written models of the same circuits. We hope that the ability to do preliminary model generation encourages more teams to try Fixture and the DaVE environment.

Chapter 7

Conclusion

Automating analog model generation is a challenging task. Early in this work, it seemed like the best solution for modeling many circuits was to treat them as an amplifier with the proper domain conversions. Unfortunately, as we focused on modeling more circuit nonidealities it became clear that this was not a sustainable solution. Taking the example of a phase blender, an ideal model is essentially an amplifier with its inputs and outputs converted to the phase domain. But the nonidealities (glitching behavior), specifics of the input space (variability between input clock phases), and difficulties in analysis (aliasing between phase blenders with a 2π difference in output delay) are all difficult to translate to the amplifier template. Rather than create amplifier template full of phase-blender-specific cases, we decided that it was best to create a dedicated template. This reinforced the conclusions of previous work on DaVE, and led to the structure of the Fixture library today. With this in mind, we still believe that the vast majority of analog circuits can be modeled with relatively few (perhaps a few dozen) templates. Additionally, templates for similar circuit types can still share code for blocks like domain converters, allowing good organization of circuit types while sharing as many modeling strategies as possible.

Although the limited number of analog circuit types allows us to reuse existing models and modeling strategies to some extent, there is a huge variety to the small quirks and additions analog designers can add to their circuits that prevents them from matching a standardized template. Much of this thesis can be summarized as finding ways to model these quirks, then generalize them as much as possible, and finally teach the tool to apply them to any circuit type. Examples of this include the many types of optional input, the ability to vector required inputs, domain translation for required and optional inputs, and the ability to accept arbitrary nonlinear optional input expressions from the user. We believe that this variety of ways to modify the templates should cover a majority of user circuits, but it is difficult to predict what circuits the tool may encounter in the future.

It takes many example circuits before one can be confident that a template can handle an unknown user circuit. In our library so far, the amplifier template has seen the most test cases and

we are confident that it can be useful to new users. Although the other templates may still need further improvement to handle all user circuits, we believe the existing templates will work for most cases. Additionally, our modular library approach - with individual templates, and individual tests within the template - allows us to add necessary functionality to the library piece by piece.

Because the library is open-source and modular, we hope that some engineers are able to add new templates and tests to fit their needs. Fixture has made template creation easier in some ways than hand-writing a new circuit characterization script by automating tasks like random sampling, measurement of optional input effects, and nonlinear parameter fitting. Still, it takes a skilled engineer to write a template because Fixture cannot automate tasks like determining the right domain for a model, ensuring that all important nonidealities are modeled, and keeping simulation time in mind. In general, if an engineer could not do the circuit analysis by hand, they could not make a new template for Fixture. In addition, even if the engineer could make the model by hand they still need some Fixture-specific knowledge about how to organize the template and test before they can contribute to the library. Still, we hope that if enough engineers become familiar with the existing templates then they will be motivated to make any new circuit models they need in the same environment.

There are still many ways Fixture itself can be improved. The tool would benefit from the ability to automatically manage Monte Carlo variations of spice and functional models. In addition, a more unified approach to timing the effects of optional inputs would make it easier to implement more tests like the phase blender glitching behavior test in the future. We hope that in the future new developers will see the value in the project and continue to add features.

Most of all, we hope that users find Fixture useful. We believe that the DaVE ecosystem is better than ever with the addition of Fixture, and engineers should take advantage of this resource to improve their functional modeling workflow. When a community begins using the tools can we find and eliminate bugs, gain confidence in the existing model library, and expand the functionality for future users.

Bibliography

[1] H.V. Deshpande, Baohong Cheng, and J.C.S. Woo. "Channel engineering for analog device design in deep submicron CMOS technology for system on chip applications". In: *IEEE Transactions on Electron Devices* 49.9 (2002), pp. 1558–1565. DOI: 10.1109/TED.2002.801435.

[2] Boris Murmann. "Digitally Assisted Analog Circuits; Fifth IEEE Dallas Circuits and Systems Workshop". In: *2006 IEEE Dallas/CAS Workshop on Design, Applications, Integration and Software*. 2006, pp. 23–30. DOI: 10.1109/DCAS.2006.321026.

[3] Xin Li, Chandramouli Kashyap, and Chris J. Myers. "Guest Editors' Introduction Challenges and Opportunities in Analog/Mixed-Signal CAD". In: *IEEE Design & Test* 33.5 (2016), pp. 5–6. DOI: 10.1109/MDAT.2016.2594182.

[4] A.N. Karanicolas, H.S. Lee, and K.L. Bacrania. "A 15 b 1 Ms/s digitally self-calibrated pipeline ADC". In: *1993 IEEE International Solid-State Circuits Conference Digest of Technical Papers*. 1993, pp. 60–61. DOI: 10.1109/ISSCC.1993.280084.

[5] B. Murmann and B.E. Boser. "A 12-bit 75-MS/s pipelined ADC using open-loop residue amplification". In: *IEEE Journal of Solid-State Circuits* 38.12 (2003), pp. 2040–2050. DOI: 10.1109/JSSC.2003.819167.

[6] E. Iroaga and B. Murmann. "A 12b, 75MS/s Pipelined ADC Using Incomplete Settling". In: *2006 Symposium on VLSI Circuits, 2006. Digest of Technical Papers*. 2006, pp. 222–223. DOI: 10.1109/VLSIC.2006.1705390.

[7] Hisakatsu Yamaguchi et al. "A 5Gb/s transceiver with an ADC-based feedforward CDR and CMA adaptive equalizer in 65nm CMOS". In: *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*. 2010, pp. 168–169. DOI: 10.1109/ISSCC.2010.5434001.

[8] C.R. Grace, P.J. Hurst, and S.H. Lewis. "A 12 b 80 MS/s pipelined ADC with bootstrapped digital calibration". In: *2004 IEEE International Solid-State Circuits Conference (IEEE Cat. No.04CH37519)*. 2004, 460–539 Vol.1. DOI: 10.1109/ISSCC.2004.1332793.

[9] Yangjin Oh and B. Murmann. "System embedded ADC calibration for OFDM receivers". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 53.8 (2006), pp. 1693–1703. DOI: 10.1109/TCSI.2006.879063.

[10] Ken Kundert and Henry Chang. "Model-based functional verification". In: *Design Automation Conference*. 2010, pp. 421–424. DOI: 10.1145/1837274.1837380.

[11] Henry Chang and Ken Kundert. "Verification of Complex Analog and RF IC Designs". In: *Proceedings of the IEEE* 95.3 (2007), pp. 622–639. DOI: 10.1109/JPROC.2006.889384.

[12] William F. Ellersick. "How to Prevent a Sick ASIC". In: *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 2022, pp. 1–6. DOI: 10.1109/HPEC55821.2022.9926305.

[13] Steven Herbst. "An Open-Source Framework for FPGA Emulation of Analog/Mixed-Singal Integrated Circuit Designs". PhD thesis. Stanford Univesity, 2021.

[14] Byong Chan Lim et al. "Digital Analog Design: Enabling Mixed-Signal System Validation". In: *IEEE Design & Test* 32.1 (2015), pp. 44–52. DOI: 10.1109/MDAT.2014.2361718.

[15] Kemal Çağlar Coşkun, Muhammad Hassan, and Rolf Drechsler. "Equivalence Checking of System-Level and SPICE-Level Models of Linear Analog Filters". In: *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. 2022, pp. 160–165. DOI: 10.1109/DDECS54261.2022.9770142.

[16] Sayandeep Sanyal et al. "CoveRT: A Coverage Reporting Tool for Analog Mixed-Signal Designs". In: *2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)*. 2020, pp. 119–124. DOI: 10.1109/VLSID49098.2020.00038.

[17] Sayandeep Sanyal et al. "The Notion of Cross Coverage in AMS Design Verification". In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2020, pp. 217–222. DOI: 10.1109/ASP-DAC47756.2020.9045131.

[18] Andreas Fürtig et al. "Novel metrics for Analog Mixed-Signal coverage". In: *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. 2017, pp. 97–102. DOI: 10.1109/DDECS.2017.7934589.

[19] Sabrina Liao and Mark Horowitz. "A Verilog piecewise-linear analog behavior model for mixed-signal validation". In: *Proceedings of the IEEE 2013 Custom Integrated Circuits Conference*. 2013, pp. 1–5. DOI: 10.1109/CICC.2013.6658461.

[20] H. El Tahawy, A. Chianale, and B. Hennion. "Functional verification of analog blocks in FIDELDO: a unified mixed-mode simulation environment". In: *1989 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1989, 2012–2015 vol.3. DOI: 10.1109/ISCAS.1989.100767.

[21] Wen Chen et al. "Challenges and Trends in Modern SoC Design Verification". In: *IEEE Design & Test* 34.5 (2017), pp. 7–22. DOI: 10.1109/MDAT.2017.2735383.

[22] Bin Wan and Xingang Wang. "Overview of commercially-available analog/RF simulation engines and design environment". In: *2014 12th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*. 2014, pp. 1–4. DOI: 10.1109/ICSICT.2014.7021256.

[23] Cadence Design Systems, Inc. *Spectre AMS Designer: Flexible mixed-signal simulation for SoCs*. URL: https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/circuit-simulation/spectre-ams-designer.html.

[24] Accellera Systems Initiative. *SystemC Analog/Mixed-Signal Extensions*. URL: https://systemc.org/overview/systemc-ams//.

[25] Martin Barnasconi. "SystemC AMS Extensions: Solving the Need for Speed". In: (Jan. 2010).

[26] T. Rizzi et al. "Comparative Analysis and Optimization of the SystemC-AMS Analog Simulation Efficiency of Resistive Crossbar Arrays". In: *2021 XXXVI Conference on Design of Circuits and Integrated Systems (DCIS)*. 2021, pp. 1–6. DOI: 10.1109/DCIS53048.2021.9666193.

[27] Scientific Analog, Inc. *xmodel: Empower SystemVerilog with Event-Driven Analog Models*. URL: https://www.scianalog.com/xmodel/.

[28] Scientific Analog, Inc. *modelzen: Auto-Extract Analog Models from Circuits*. URL: https://www.scianalog.com/modelzen/.

[29] Scientific Analog, Inc. *glister: Model Circuits in Schematics without Writing Codes*. URL: https://www.scianalog.com/glister/.

[30] Ji-Eun Jang et al. "True event-driven simulation of analog/mixed-signal behaviors in SystemVerilog: A decision-feedback equalizing (DFE) receiver example". In: *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference*. 2012, pp. 1–4. DOI: 10.1109/CICC.2012.6330558.

[31] Yoontaek Lee, Jeongyeol Kwon, and Jaeha Kim. "Power Loss Analysis of Switched-mode Converter Circuits in XMODEL". In: *International Technical Conference on Circuits/Systems, Computers and Communications*. 2016, pp. 609–612. DOI: 10.34385/proc.61.5174.

[32] Ji-Eun Jang, Si-Jung Yang, and Jaeha Kim. "Event-driven simulation of Volterra series models in SystemVerilog". In: *Proceedings of the IEEE 2013 Custom Integrated Circuits Conference*. 2013, pp. 1–4. DOI: 10.1109/CICC.2013.6658460.

[33] Lei Yang and C.-J.R. Shi. "FROSTY: a fast hierarchy extractor for industrial CMOS circuits". In: *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486)*. 2003, pp. 741–746. DOI: 10.1109/ICCAD.2003.159759.

[34] Seyoung Kim and Jaeha Kim. "An Equivalent Modeling Approach for High-Density DRAM Array System-Level Design-Space Exploration in SystemVerilog". In: Apr. 2021.

[35] Hyun-Sek Lukas Lee et al. "Automated generation of hybrid system models for reachability analysis of nonlinear analog circuits". In: *The 20th Asia and South Pacific Design Automation Conference*. 2015, pp. 725–730. DOI: 10.1109/ASPDAC.2015.7059096.

[36] J.R. Burch et al. "Sequential circuit verification using symbolic model checking". In: *27th ACM/IEEE Design Automation Conference*. 1990, pp. 46–51. DOI: 10.1109/DAC.1990.114827.

[37] Cadence Design Systems, Inc. *AMS Design and Model Validation User Guide IC6.1.8*. URL: http://support.cadence.com.

[38] Amandeep Singh and Peng Li. "On behavioral model equivalence checking for large analog/mixed signal systems". In: *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2010, pp. 55–61. DOI: 10.1109/ICCAD.2010.5651402.

[39] Byong Chan Lim. *StanfordVLSI: DaVE*. URL: https://github.com/StanfordVLSI/DaVE.

[40] Byong Chan Lim. "Model Validation of Mixed-Signal Systems". PhD thesis. Stanford Univesity, 2012.

[41] Byong Chan Lim and Mark Horowitz. "An Analog Model Template Library: Simplifying Chip-Level, Mixed-Signal Design Verification". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.1 (2019), pp. 193–204. DOI: 10.1109/TVLSI.2018.2873387.

[42] Mark Horowitz et al. "Fortifying analog models with equivalence checking and coverage analysis". In: *Design Automation Conference*. 2010, pp. 425–430. DOI: 10.1145/1837274.1837381.

[43] Jaeha Kim, Kevin D. Jones, and Mark A. Horowitz. "Variable domain transformation for linear PAC analysis of mixed-signal systems". In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. 2007, pp. 887–894. DOI: 10.1109/ICCAD.2007.4397376.

[44] Byong Chan Lim, Jaeha Kim, and Mark A. Horowitz. "An efficient test vector generation for checking analog/mixed-signal functional models". In: *Design Automation Conference*. 2010, pp. 767–772.

[45] Byong Chan Lim and Mark Horowitz. "Error Control and Limit Cycle Elimination in Event-Driven Piecewise Linear Analog Functional Models". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 63.1 (2016), pp. 23–33. DOI: 10.1109/TCSI.2015.2512699.

[46] Michał Rewieński. "A Perspective on Fast-SPICE Simulation Technology". In: *Simulation and Verification of Electronic and Biological Systems*. Ed. by Peng Li, Luís Miguel Silveira, and Peter Feldmann. Dordrecht: Springer Netherlands, 2011, pp. 23–42. ISBN: 978-94-007-0149-6. DOI: 10.1007/978-94-007-0149-6_2. URL: https://doi.org/10.1007/978-94-007-0149-6_2.

[47] *Liberty User Guides and Reference Manual Suite Version 2017.06*. Synopsys, Inc. 2017.

[48] Quan Hu, Lijuan Yang, and Fengyi Huang. "A 100–170MHz fully-differential Sallen-Key 6th-order low-pass filter for wideband wireless communication". In: *2016 International Conference on Integrated Circuits and Microsystems (ICICM)*. 2016, pp. 324–328. DOI: 10.1109/ICAM.2016.7813617.

[49] Aaron Meurer et al. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[50] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. ISSN: 1476-4687. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[51] Sung-Jin Kim et al. "Open-Source Synthesizable Analog Blocks for High-Speed Link Designs: 20-GS/s 5b ENOB Analog-to-Digital Converter and 5-GHz Phase Interpolator". In: *2020 IEEE Symposium on VLSI Circuits*. 2020, pp. 1–2. DOI: 10.1109/VLSICircuits18222.2020.9162800.

[52] Sung-Jin Kim et al. "20-GS/s 8-bit Analog-to-Digital Converter and 5-GHz Phase Interpolator for Open-Source Synthesizable High-Speed Link Applications". In: *IEEE Solid-State Circuits Letters* 3 (2020), pp. 518–521. DOI: 10.1109/LSSC.2020.3037823.

[53] M. D. McKay, R. J. Beckman, and W. J. Conover. "Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code". In: *Technometrics* 21.2 (1979), pp. 239–245. DOI: 10.1080/00401706.1979.10489755. eprint: https://doi.org/10.1080/00401706.1979.10489755. URL: https://doi.org/10.1080/00401706.1979.10489755.

[54] Krzysztof Choromanski et al. "Unifying Orthogonal Monte Carlo Methods". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 1203–1212. URL: https://proceedings.mlr.press/v97/choromanski19a.html.

[55] Lenny Truong et al. *fault: A Python Embedded Domain-Specific Language For Metaprogramming Portable Hardware Verification Components*. 2020. arXiv: 2006.11669 [cs.SE].

[56] Rick Bahr et al. "Creating an Agile Hardware Design Flow". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218553.

[57] ngspice Developers. *ngspice: Mixed Mode - Mixed Level Circuit Simulator*. URL: https://ngspice.sourceforge.io/index.html.

[58] Aleksandar Donev, Salvatore Torquato, and Frank H. Stillinger. "Neighbor list collision-driven molecular dynamics simulation for nonspherical hard particles. I. Algorithmic details". In: *Journal of Computational Physics* 202.2 (2005), pp. 737–764. ISSN: 0021-9991.

[59] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[60] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.