CIRCUITBOOK: A FRAMEWORK FOR ANALOG DESIGN REUSE

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

James Mao
May 2013

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.

Ada Poon

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.

Bruce Wooley

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

Abstract

Analog IC design tools have not changed much during the past few decades. While the models and simulation methods have greatly improved in accuracy and performance, analog design still relies on manually constructed schematics and layouts. Each design needs to carry its own test routine, and the quality of the entire system depends on this custom-constructed test routine. It is common for circuits to be tested in an ad-hoc manner through some combination of SPICE decks, MATLAB calculations, and perl scripts. This test collateral is often brittle (i.e., tightly coupled to the process, circuit, and simulation environment) and archived without sufficient documentation. As a result, it is usually easier to recreate the test frame when the circuit is reused in the future.

Languages like OCEAN try to address this by providing a common language that both configures the simulation environment and performs calculations. While this reduces the number of files required for a simulation and reduces the need for documentation, it does not solve the problem of tight coupling. Reuse of OCEAN scripts generally means copy-and-paste, and this duplication of code makes the test routines more difficult to debug later. We see an opportunity to improve the productivity of analog design by raising the abstraction level, at least for test construction, used by analog designers. In this work, we present the CircuitBook test framework and repository that complement existing analog design flows.

Our test framework is a set of Python libraries that allow high-level specification of analog tests and an associated tool chain that executes these tests. Circuits and tests are defined against an hierarchical tree of interfaces. These common interfaces allow the reuse of tests across different circuits as well as enable faster prototyping of circuits and tests. Tests defined using the CircuitBook test framework separate the simulation directives from the results analysis. This separation of concerns avoids code duplication by separating the reusable parts of the tests from the environment-specific parts.

The CircuitBook repository stores circuits, tests, and simulation results to allow designers to leverage existing circuits and tests in new designs. The objects in this repository can browsed via the interface hierarchy or through property tags of the tests. We leverage the high-level nature of the test framework and automatically generate property tags by parsing the test structure.

Acknowledgements

I am grateful for the generous mentoring and support that my advisor, Prof. Mark Horowitz, has provided over the years.

I would also like to thank for Prof. Ada Poon and Prof. Bruce Wooley for serving on my Orals committee and reading this thesis, as well as Prof. Roger Howe for chairing my Orals committee.

This thesis and the work described within contain the ideas and refinements of many, including Prof. Elad Alon and Prof. Jaeha Kim, Metha Jeeradit, and Byong Chan Lim. This list is by no means exhaustive.

This research was made possible by the generous support of Sony through the Stanford Graduate Fellowships Program and by the corporate sponsors of the Rethinking Analog Design initiative.

Finally, this thesis would not be nearly as fun to read if it were not for my wife Lillian's mastery of the English language.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

We live in the age of the mixed-signal system-on-chip (SOC). Over 70% of new designs in modern technology processes are now mixed-signal (i.e., at least 20% of the die area is analog), primarily as a result of the need to integrate RF, high-speed I/O, and multiple clocks on one chip. While the number of analog transistors on a mixed-signal IC is small compared to the number of digital transistors, analog and digital circuits require roughly the same amount of design effort and cause similar numbers of re-spins due to design mistakes[1]. One reason for this difference in design productivity is that analog and digital CAD tools are quite different today. Most modern digital design is done by writing some code in a high-level hardware description language (HDL) and then running that code through synthesis and place and route tools. Although these digital flows may sometimes require magic incantations to work, the tools automate many common tasks and allow the designer to focus on the overall systems and the details of the trouble areas. On the other hand, analog design is primarily done using manually constructed schematics and layouts. There is some automation in terms of low level layout cell generators (e.g., PCells) but this automation works at the level of macros (i.e., repeat a set of steps).

Both analog and digital chips started with full custom design processes (i.e., transistor level designs with SPICE simulations and custom layout) in the 1960s and 1970s. Digital tools quickly evolved over the past 40 years, but analog tools did not change as much. As a result, today, analog design is done at a much lower level than digital design. This disparity provides us an opportunity to improve the productivity of analog designers by raising the level of abstraction they use to design.

In order to understand why analog design tools have lagged behind digital tools, Chapter 2 describes the analog circuit design process and the challenges faced by analog designers. From examining this processes, we note that many of these challenges, and most of the design time, is focused on validation. Tests play an important part in the circuit design process, but test collateral often is not reused due to the nature of ad-hoc tests. Reuse of ad-hoc tests is hard because these tests intermingle reusable pieces and non-reusable pieces. By breaking up tests into the basic constituent pieces, we see that there is a way of structuring tests that will facilitate reuse.

In Chapter 3, we examine the basic components of a test (i.e., circuit representations, simulation directives, stimuli, measurement, and results analysis routines) and the interaction between these components. We leverage the natural interfaces between the various pieces of a test to define formal interfaces. By using these interfaces, we are able to better segregate reusable test components from less reusable pieces. We define circuit interfaces that completely separate circuits from the tests that operate on them. These interfaces contain a set of physical and informational ports that allow the design of circuit-independent tests. In addition, these interfaces are hierarchical and allow tests to be pulled up (i.e., rewritten to work on a more general class of circuits). Similarly, we also present our abstractions for stimuli and measurement components. Once we establish these basic elements, we look at how they allow us to construct a hierarchical repository for organizing circuits and tests.

Chapter 4 looks at the details of our test framework. It starts by discussing the way circuits and interfaces are specified. Then, it examines the `SimulationRun` scripts, which represent the reusable cores of tests written in our test framework, and how these scripts handle stimulus generation, simulation execution, measurement, and reporting. Next, we look at how `TestRun` and `TestSequence` capture the less reusable parts of tests and how these tests are executed by our framework. Finally, we examine how to extract metadata from test collateral for organization purposes.

Chapter 2

Analog Circuit Design Process

Analog circuit design is the process of connecting devices to each other such that the resulting circuits meet a set of specifications while minimizing other metrics, such as area, power, or yield loss. This process produces a circuit schematic and the corresponding physical layout. The overall process is shown in Figure 2.1. In industry, the specifications are generally created by examining the intersections of customer demand, process capabilities, design expertise, and business goals. Academic specifications may be more exploratory in nature, investigating new architectures or methodologies. Not all specifications are explicitly stated; design groups may have internal, unwritten rules reflecting best practices. The goal of a circuit designer is to create a circuit that satisfies the specifications provided, within a certain time-to-market window.

To accomplish this goal, the designer first performs some exploration of the design space using rough models to pick a circuit topology; this is typically done via a combination of hand calculations and tools such as MATLAB[2], Simulink[3], or Mathcad[4]. Once a topology candidate is selected, a circuit schematic is created using a schematic capture tool (e.g., Cadence Virtuoso Analog Design Environment[5]). Values for the various circuit elements are initially determined by some rough back-of-the-envelope calculations using analytic design equations. The circuit schematic is then run through one or more circuit simulations on a tool such as SPICE[6] or

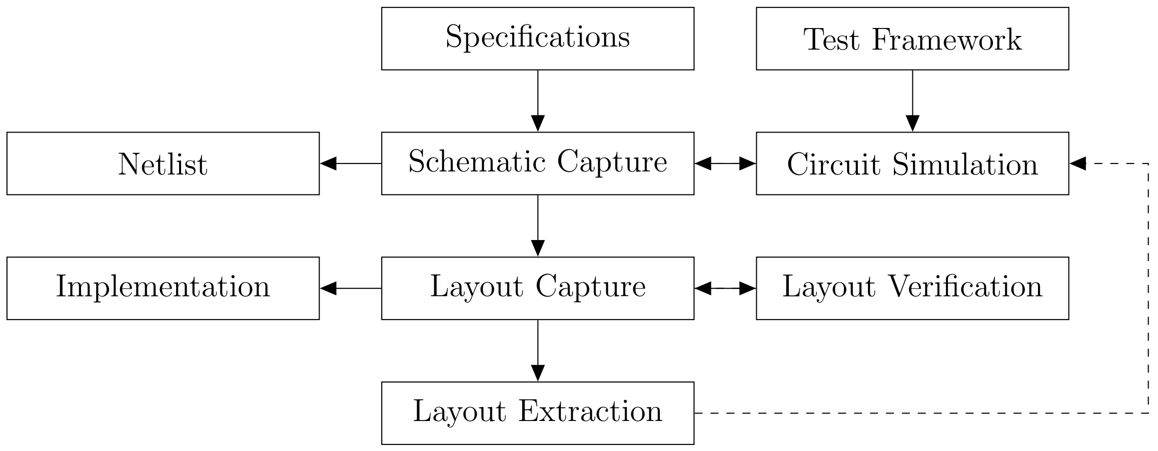


Figure 2.1: Overall View of Analog Circuit Design Task

Spectre[7]. Some post-processing is done to extract higher-level performance characteristics. This combination of simulation and measurement tests whether the circuit meets the specifications and can also be used to estimate optimization metrics.

Design is an iterative process. The test results from one iteration are used to modify the circuit design and the updated schematic is resimulated. The process is repeated until the circuit meets design goals; once that happens, the design is sent to layout, which is the process of converting circuit elements to geometric shapes for the fabrication process. Layouts are usually created using a tool such as Virtuoso Layout Editor[5] from Cadence or Galaxy Custom Designer[8] from Synopsys. These layouts are run through a verification step that performs a layout versus schematic (LVS) comparison and design rule checks (DRC). LVS ensures that the connectivity of nets in the layout matches the connectivity of nets in the schematic. DRC ensures that the layout meets the foundry's requirements for manufacturing the design in a particular technology process. Once the layout passes LVS and DRC, the resistance and capacitance of wires used in layout are extracted and back-annotated into the schematic. Finally, the entire design is tested again with the more accurate schematics to validate the design before shipping.

In the remainder of this chapter, we will explore the opportunity to improve the circuit design process through an improved infrastructure for evaluating (testing) circuit performance. To understand why we focus on testing, we first survey the

Faster Iterations	**Fewer Iterations**
Faster simulators – AFS, UltraSim	Circuit reuse – VCME, BAG
New analysis methods – PAC	Design methodology – g_m/I_D
Better tool integration – Virtuoso Suite	Better device models – BSIM4

Table 2.1: Landscape of Design Productivity Improvements

landscape of recent productivity improvements to the circuit design process. Next, we will examine the role of tests in the circuit design process, showing the critical role that it plays and the ad-hoc way in which tests are currently constructed. Finally, we will present our approach to creating reusable tests.

2.1 Improvements in Designer Productivity

Since circuit design is an iterative process, there are two fundamentally different approaches to improve designer productivity in the existing design flow. We can either make each iteration faster or we can reduce the number of design iterations. Some examples of each approach are shown in Table 2.1.

2.1.1 Faster Iterations

Often, electronic design automation (EDA) vendors supplying computer aided design (CAD) tools develop improved tools to make each design iteration faster. In terms of circuit simulation, this has led to the development of tools such as Analog Fast-SPICE (AFS) from Berkeley Design Automation or Cadence's Virtuoso UltraSim. These modern circuit simulators combine fast solvers designed for modern multi-core computing platforms with hierarchical decomposition and variable time step control to provide orders of magnitude faster simulation than traditional SPICE with the same or only slightly worse accuracy. For example, AFS is approximately 5-10x faster than SPICE with the same accuracy ($< 0.1\%$)[1] and UltraSim can be up to 100x faster than SPICE with accuracy in the range of 1-5%[9]. In addition, AFS can handle larger circuits (i.e., 10x the number of elements) than SPICE[1]. Recent research shows the potential for even greater speedups using techniques such as applying the latency insertion method to general circuit simulation[10]. Faster simulation times reduce the time needed for circuit evaluation, which makes each iteration faster.

In addition to performing better on traditional simulation modes (i.e, DC, AC, and transient analyses), modern circuit simulators like Spectre support new simulation modes such as periodic small signal analysis, which can greatly reduce the amount of computation required to evaluate the complex behavior of periodic circuits such as phase-locked loops (PLLs). Such periodic circuits often have one or more inputs with a large periodic signal (e.g., clocks). Periodic small signal analysis modes, including periodic AC (PAC), periodic steady state (PSS), and periodic noise (Pnoise), work by first computing the circuit response with only the periodic inputs and using that solution as an operating point for future analysis[11]. This can greatly reduce simulation time by separating the large periodic inputs that are non-linear from the linear input of interest.

Similarly, tools such as Cadence Virtuoso Layout Editor and Synopsys Custom Designer help improve productivity in the layout phase of the circuit design process through better user interfaces and better handling of hierarchy. EDA industry consolidation has also led to better integration between tools which shortens total iteration times by reducing friction between tools (e.g., Synopsys' Galaxy Implementation Platform or Cadence's Virtuoso Suite). Tool integration is a combination of features that improve productivity: user interface integration (e.g., unified look-and-feel makes it easy to learn related tools), data integration (e.g., common data formats that eliminate the need for translation and processing), and control integration (e.g., tools communicate with each other)[12].

2.1.2 Fewer Iterations

In addition to improving the speed of each design iteration, there have been efforts in industry and academia to reduce the number of iterations required. These efforts mainly lie along one of two directions: better modeling and reuse.

The physical devices in today's deep submicron technologies are becoming increasingly complicated. Square law models used in the past for hand analysis are no longer sufficiently accurate. Relying on those antiquated models results in large differences between predicted behavior and simulated results, which then require more design iterations to correct. Such a "SPICE monkey" approach is clearly unproductive.

There have been improvements in design methodology that address this issue. The g_m/I_D methodology proposed by Silveira, Flandre, and Jespers[13] replaces square law models with lookup tables of simulated or measured device characteristics. This naturally leads to an increased use of computer solvers in the design process (i.e., use MATLAB with g_m/I_D curves derived from simulations in lieu of hand calculations with square law models). For example, when Flandre et al. uses the g_m/I_D methodology to develop gain-boosted regulated-cascode operational transconductance amplifiers[14], they use ISAAC[15], a symbolic simulator for analog circuits, in combination with MATLAB to study the small-signal behavior. Such methodology changes address the discrepancy between design equations and simulation models. Refining transistor simulation models (e.g., UC Berkeley Device Group's efforts on BSIM4) to minimize the difference between simulation models and fabricated devices has also been critical. In addition, there has been work on system-level modeling of mixed-signal systems for synthesis[16, 17] and verification[18, 19]. These ideas allow designers to work at a higher abstraction level when architecting mixed-signal systems. Combined with the g_m/I_D methodology and improved device level modeling, these abstraction techniques allow designers to work at the appropriate level to maximize productivity while maintaining accuracy (i.e., building designs that work according to the models). When successful, the resulting circuits are closer to meeting the specifications and require fewer iterations.

Design reuse goes one step further and tries to eliminate some design iterations completely. Even in a cutting-edge mixed-signal design, not all cells require novel techniques; many cells in such a design have been designed before (e.g., LDO regulator, biasing circuits, simple amplifiers, etc.). Design reuse can reduce or eliminate the effort to design some of these supporting analog blocks as well as increase the reliability of the resulting design.

Design reuse can be achieved in multiple ways, with different trade-offs between effort needed to prepare the reusable collateral and effort needed to customize the reusable collateral for a particular application. At one extreme, reuse can be achieved by simply archiving old designs (e.g., a tar archive of all design files). At the other, a design generator is created, which generates the desired circuit algorithmically.

Archiving a design requires only a small amount of effort in preparing the collateral, but effectively using the archived designs is a challenge. Often the desired circuitry has a slightly different specification, or uses a different technology than the archived design. Thus, using the archived design is an instance of the general design migration / porting problem – producing netlists and layouts in a new technology from existing netlists and layouts in an old technology while maintaining or improving the performance of the design. The port needs to be faithful to the old design so that new problems do not appear. Typically, the source design has been successfully fabricated and tested, and an ideal migration process would preserve this correctness in the new design.

Migrating digital designs captured in HDL between technology nodes is mostly a matter of porting the design library and is relatively well understood[20, 21]. Meanwhile, for analog or mixed-signal designs, porting remains an area of continuing investigation: In 1998, Funaba et al. showed a manual, optimization-based method for porting circuit designs. Francken and Gielen presented a semi-automatic method of porting analog circuits and layouts across technologies[22] in 1999 with initial sizes calculated by scaling and subsequently tuned via qualitative reasoning[23]; in this method, layout migration starts by scaling the floorplan with semi-automatic routing – automatic routing at the lower levels with the top level manually routed. In early 2000s, Neolinear developed a set of commercial tools (i.e., NeoCircuit and NeoCell) that automatically sizes schematics based on designer supplied test benches and constraints[24]. In 2006, Hammouda et al. presented Chameleon ART[25], a knowledge-based design migration tool that identifies the structure of certain analog blocks and uses the sizing rules methodology[26] developed by Massier, Graeb, and Schlichtmann to set device sizes; layout migration in Chameleon ART is achieved through scaling, followed by constraint based compaction. Finally, Weng et al. presented a methodology for migrating layouts in 2011 by extracting analog layout constraints and generating multiple layouts to allow the user to choose among various aspect ratios, an option not available with scale and compact layout migration techniques[27]. As the porting tools get more powerful, it becomes increasingly important that the critical constraints are codified so that the tools obey them.

While there has been progress made in porting designs between technologies, these ideas only address a part of the problem. Simply resizing devices for a new technology or application is often insufficient to achieve desired design metrics. The architecture of the system or one of the sub-blocks may need to be changed. Making these types of changes generally requires some design knowledge in addition to optimization, which brings us to the other end of the reuse spectrum – generating circuits algorithmically instead of trying to adapt design collateral. Since the late 1980s, there have been many research tools focusing on synthesis of analog building blocks, such as IDAC[28], OPASYN[29], BLADES[30], DSYN[31], CADICS[32], OASYS[33], ASTRX/OBLX[34], and Anaconda[35]. In contrast to simply reusing old designs, this approach requires lots of initial effort, but provides adaptation for each application with little user intervention. The tools combine topology selection, parametric optimization, and automatic layout generation to produce schematics and layouts for analog building blocks. While the various tools introduce and leverage different techniques for topology selection (i.e., heuristic or algorithmic), device sizing, parameter estimation, or layout generation, the general methodology is similar. Figure 2.2 shows this general strategy.

Earlier tools, developed around 1990, (e.g., IDAC, OPASYN) relied heavily on designer knowledge encapsulated in circuit topology libraries and design plans, a program written by the user, to guide the synthesis process. These tools also used design equations for performance estimation instead of circuit simulations due to computational limitations. The initial effort of creating accurate design plans limited the application of these tools[36]. As available computing power increased, analog synthesis tools began to replace some knowledge-based topology selection and design plans with algorithmic searches. This reduced the initial investment required to operate

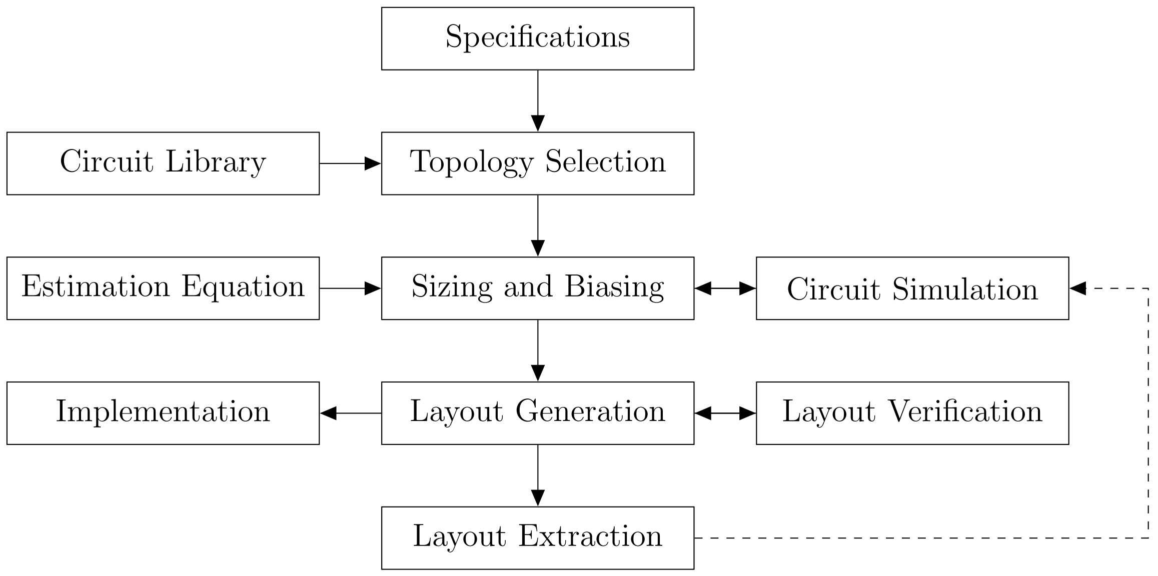


Figure 2.2: Typical Analog Synthesis Process

such a system by reducing or eliminating the reliance on a design plan. The focus of research shifted from building specific analog generators (e.g., earlier tools focused on building data converters and their building blocks[37]) to a platform approach to design reuse (i.e., design environments that enable circuit reuse by storing parameterized circuits). Examples of such tools include AMGIE[38], BAG[39], and VCME (Virtuoso Characterization and Modeling Environment from Cadence).

The movement to general optimization frameworks again makes it essential to capture all of the constraints on the design. Mathematical optimization has no "common sense," so all implicit design rules need to be made explicit. However, these tools do not provide a methodology to create the necessary constraints. The constraints are generally codified as a test script, to measure the circuit's performance, but these scripts are created in an ad-hoc manner. This thesis addresses how to better capture and archive these tests.

2.2 The Role of Tests

Reviewing the prior work makes clear the critical role that tests play in analog design and the even more important role they will play if analog cell generators become popular. Yet at the same time, there seems to be a lack of focus on the process of creating and reusing this test collateral for circuits. Improving the productivity of test construction has significant potential, since testing is integral to both the way design is done today and possible paths to future automation.

Tests are created and executed to evaluate operational performance and verify that the circuit meets certain invariants (e.g., current source devices in saturation). The circuit design process is not complete until each desired specification is measured through simulation. Testing also acts as a form of institutional memory. It is used to capture the knowledge of mistakes that have caused respins or delays in the past and tries to avoid that same class of errors in future designs by creating a specification that prevents that kind of error from reoccurring. For example, if a chip comes back with excess output noise and that is traced back to noise coupling on the bias line, a test would be created to ensure that this coupling is small on future designs. Requiring a certain test to pass is a concise way of transferring a set of learnings to other members of the team. As a result, every member of the team should contribute checks to ensure that their fears about a design do not occur in the current implementation. These checks would be in the form of tests that run on the individual blocks or the entire system.

Conceptually, a test is composed of four parts: a test bench, test vectors, simulation directives, and result analyses. The test bench contains the set of all additional elements added to the circuit, often called a device under test (DUT) in test contexts, for simulation purposes. A typical test bench contains power supplies (e.g. ideal voltage sources or complex models representing the power distribution network), input sources (e.g., sine waves or piecewise linear functions), output loading, and measurements (i.e., measurement commands written in SPICE to extract the parameters of interest). Parts of the test bench, such as the input sources, require configuration that changes between runs. A test vector is a set of configuration values for a single run and the expected outputs. For example, a data converter needs to be tested with different input sequences to ensure that there is no hysteresis. In that case, a test vector may contain an input sequence and the expected measurement values or ranges. Simulation directives tell the simulator (e.g. SPICE, or Spectre) which analyses to perform, how long to simulate, and the desired accuracy. The output of the simulator is run through a set of result analysis routines that perform the post-processing necessary to extract the parameters of interest from the measured values. For example, a result analysis script calculates various jitter metrics (e.g., absolute jitter, period jitter, and cycle-to-cycle jitter) from a list of zero-crossing times produced by simulating the test bench.

Writing this test collateral requires a significant amount of effort due to the large number of specifications to extract, and the test itself represents significant design knowledge. Given the value of these tests, it is wasteful if they are not reused. However, reuse of ad-hoc tests can problematic. The next section looks at this issue in more detail to understand what makes reusing tests hard.

2.3 Ad-hoc Testing

Ad-hoc testing is, as the name implies, ad-hoc; tests from different designers can be constructed very differently, with different programming languages, different simulators, and different post-processing techniques. The resulting diversity makes reuse challenging for any application.

Verifying the correct operation of an analog circuit implementation typically requires four kinds of source code: code provided to configure the simulator, code to perform measurements (either during simulation or afterwards), code to extract desired parameters from the measurement results and perform any post-processing needed, and code to tie everything together. When a test is created, each of these components can be expressed in a multitude of languages and styles based on designer preferences, skills, and aesthetics.

A typical test constructed in this ad-hoc manner contains a variety of scripts written in different programming languages. At the top level, there are shell scripts that control the test flow, set up the simulation environment, and execute the simulations. Measurement is generally done using a combination of hand-coded `.measure` statements in the simulators, some selections in a GUI (e.g., Virtuoso ADE XL from Cadence), and some post-processing Perl code to prepare the output for analysis. Analysis of the measured results is often performed using a combination MATLAB scripts and Excel spreadsheets.

Ad-hoc test collateral, as described above, is a complex software project written in multiple languages. But even if there were a single unified language, some issues would remain. Many designers see testing as an extension of the design work they are already used to doing; they do not see it as a software project. More importantly, their organizations do not see it as a software project, and, as a result, many practices common in software engineering are not applied. For example, it is common for software development organizations to enforce code standards, code reviews, and version control systems. Code standards differ between organizations but generally include guidelines on syntax, indentation, formatting, and nomenclature to make code consistent and easier to understand. When it comes to testing, however, circuit design teams may review the list of checks being run on a circuit, but may not actually review the test code. In addition, while most design groups today use a version control for their designs, these tools, sometimes a part of larger configuration management suites, are geared toward the maintenance of design databases and may not be ideal for source code, even if used for tests. Without these software engineering practices, test collateral may not be traceable. That is, it may not be possible to identify the test code responsible for producing a particular result. Old versions of test code may be lost; bugs may get fixed and reintroduced later.

Ad-hoc test collateral is often simply an archive of the working directory. While there is nothing inherently problematic with this approach, it is often accompanied by several practices that should be avoided. First, problems arise when tests and the files containing the tests are inconsistently named. For example, informal attempts at versioning might involve duplicating certain files (e.g., `test_v1.m`, `test_v2.m`, etc), but often these efforts do not prevent files ending up named like `new_test.m` and `new_test_fixed.m`. As such files proliferate, it becomes more and more difficult to figure out which file to run. Second, in an ad-hoc approach, code reuse is generally achieved through copy-and-paste. Often code is duplicated and slightly modified rather than reused as a whole to avoid having to identify a common interface or to avoid limiting the scope of internal variables. But when code is shared between various tests (e.g., a startup sequence), it should ideally exist exactly once so it is not possible for it to become out-of-sync. Multiple versions of duplicated code means that it is harder to fix bugs later, as a single bug must be fixed in multiple locations. Thus, while the test writer has avoided the work of maintaining a single, globally applicable version of the code, extra effort becomes necessary elsewhere.

Sometimes these problems are not entirely the fault of the test writer; they may stem in part from a limitation of the environment. For example, each MATLAB script (i.e., `.m` file) can only contain a single function[40, p. 14-22]. This makes it inconvenient to separate code into too many discrete functions. Finding a balance between MATLAB script proliferation and code duplication may not be easy or a priority for test writers. The problem is compounded when test writers apply informal version control, as changing a file name changes the name of the function, which must then be updated in all routines in which the function is called.

These common characteristics of ad-hoc test creation, which make code segments difficult to reuse, also contribute to unexpected effects as code is changed and updated. For example, programmers often find it convenient for all variables to have global scope. Global scoping means that one does not have to pass many variables into a function or a copy-and-pasted code segment. Then, there is a temptation to reuse certain pre-calculated results (i.e., premature optimization). An example of this is the use of loop indexes. We will present this example using MATLAB but similar problems exist in most dynamic languages. Suppose we have an array, `A`, and we need to iterate through that array. A common way of doing this is to iterate through this array is by enumerating all the indexes.

```
for i = 1:size(A, 1)
  ...
end
```

Usually this array may need to be iterated through many times, so it may be tempting to pre-compute the length of the array.

```
arr_len = size(A, 1)

for i = 1:arr_len
  ...
end


...


for i = 1:arr_len
  ...
end
```

This works fine as long as `arr_len` is not changed out of sync with the array `A`. Violating that covenant can result in subtle errors. Usually errors do not occur when the code is first written, since all of the dependencies are fresh. Over time, or in different reused contexts, errors pop up. These errors can be avoided with some best practices[1] but can be difficult to debug.

On a small contained project that involves only a few people over a few months, the issues described above may be manageable. Indeed, many circuit designs may start off like that. However, designs may get ported to new technology nodes or derivatives may be introduced. Often these ports or derivatives may be assigned to different, often more junior, people not familiar with the original design. That is when serious problems can surface. The original designer used their skill set to cobble together a test that worked. The new designer may not have the same skill set, for a variety of reasons other than experience. For example, older designers may be more familiar with Perl while newer designers may be more familiar with Python.

[1]Typically, it is good to use iterate through arrays without using indexes (e.g., `for v in A`). If that is not possible, then it is usually a good idea to compute the length of the array in each `for` loop. If the length function is slow enough to cause any noticeable performance degradation, then there is a more fundamental data structure problem.

Similarly, some designers prefer HSPICE while others favor Spectre. This diversity of skill sets is beneficial to teams as a whole because they can tackle a bigger set of problems. However, it causes problems during the handover of test collateral.

This accumulation of small issues is often referred to as “technical debt” in software engineering literature. Brown et al. provides an overview of the philosophical discussion surrounding the management of technical debt[41].

2.4 CircuitBook, an Approach to Reusable Tests

To address these reuse issues, we built CircuitBook, which consists of a test framework and associated repository. CircuitBook improves circuit and test reuse by taking a test-driven approach to design that not only enables faster design with increased predictability for individual blocks, but also increases the chances of success of the overall design. A repository of reusable test collateral ensures that the different circuit candidates for a block are evaluated using the same test procedure, which drives predictability. Designs often fail due to an error that has been seen before. When an error is found in a silicon revision, tests can be created to cover that class of errors. If these tests are run on future revisions and derivatives, design success rates are increased.

The rest of this section discusses the theoretical underpinnings of the framework and how it addresses the issues associated with ad-hoc testing. The core concepts of the CircuitBook test framework are presented in Chapter 3. Chapter 4 then provides a detailed discussion of how each component of the framework works.

2.4.1 A Solution Stack for Analog Testing

As discussed previously in Section 2.2 and Section 2.3, test development in today’s design and verification environments looks like software development. The process of testing analog blocks is similar to unit testing in software: take a black box function with inputs and outputs and determine whether that function works as expected. The stimulus and post-processing routines are software programs that look like parts of software unit tests. This similarity inspires us to evaluate and adopt techniques from software engineering for the verification of mixed-signal systems.

Before addressing our testing problem, it will be helpful to understand why software is often developed on common stacks of software tools — a solution stack. For example, many web applications are built on top of LAMP[42], a solution stack comprised of the Linux operating system, the Apache web server, the MySQL database, and the PHP programming language. Even if this LAMP stack is not the best set of technologies for a desired application, it may still be efficient to choose LAMP for practical reasons: it is well maintained and there are lots of knowledgeable programmers. An esoteric combination of technologies may seem to be a better framework, but can lead to previously undiscovered errors due to a untested interaction. It also may be difficult to hire engineers who understand an uncommon set of technologies. LAMP may be popular because it is popular.

In contrast, ad-hoc tests are crafted by individual designers using clever combinations of tools from the tools they know. As mentioned previously, testing circuits often requires a variety of programming languages to handle the different parts of testing (e.g., flow control, measurement, simulation, post-processing, etc.). To avoid the resulting problems, CircuitBook is a set of Python libraries that allows high-level description of simulation and measurements and is designed to serve as a solution stack for writing reusable, structured tests for mixed-signal circuits. We provide libraries that allow users to describe tests in a high-level way, much like how the MATLAB Aerospace Blockset[43] allows users to describe aerospace analysis problems.

A major advantage of the CircuitBook test framework is that the user is able to describe all of the parts of a test (test bench, test vectors, simulation directives, and result analyses, as identified in Section 2.2) in a single programming language.

Our main objective is to provide a simple solution stack that is optimized for analog testing. This mantra serves as the guiding principle behind the development of CircuitBook. Whenever possible, we opt for simplicity over complexity and specificity over generality. In terms of simplicity, we mean simplicity for the user (i.e., the framework itself is actually quite complex and uses a variety of programming tricks to appear simple to the user). We chose Python as the language of the framework for its versatility, which allows the user to write all of the various parts of the test in Python. Solution stacks are often very general (e.g., the LAMP stack mentioned earlier can be used for various classes of web-based applications, ranging from static sites to heavily data-driven services) because that generality makes it accessible to a larger audience and helps drive adoption. However, generality is not an advantage for the CircuitBook test framework, which we want to customize to the task of designing reusable tests. For example, generality means that many niche routines are independently rewritten by different users of the system, which can lead to incorrect implementations. We are able to avoid this generality by decomposing the testing problem and then providing a set of basic elements that is tailored to each of the resulting problem domains. This decomposition is discussed in Section 3.1.

Other Solution Stacks

Looking at the tools that are currently being used, there are several other candidate solution stacks for test development. A comparison of CircuitBook and two other solution stacks is shown in Table 2.2.

The most obvious alternative solution stack is a MATLAB, Python, and SPICE stack. This is close to what many designers are already using for verification tools. MATLAB is selected due to its ubiquity. Python is a versatile glue language that can perform the necessary program instantiations, numeric calculations, and data manipulation. Perl may also be a valid choice here, but the distinction is not that important; the key is to standardize around a single programming language. Other programming languages such as Ruby and shell scripts are less suitable: Ruby is not widely used in scientific computing, so its computation libraries are not as well-supported; and it is difficult to manipulate complex data structures in shell scripts, and shells may be different between various environments (e.g., workstation versus compute farm). For the last piece of the solution stack, SPICE is a good common simulation platform because most circuit design tools work with SPICE netlists. The disadvantages of this approach are the number of different programming languages involved (e.g., MATLAB, shell scripts, Python) and the lack of structure (i.e., what should be done in the Python code versus MATLAB code). CircuitBook takes a similar approach but tries to eliminate these disadvantages (i.e., single language, structured classes).

We can also imagine an alternative solution stack based on Cadence tools. Virtuoso can be scripted using SKILL[44], a dialect of Lisp. OCEAN[45] (Open Command Environment for Analysis) from Cadence is a framework for SKILL that allows a user to set up simulations, run those simulations, and analyze the results. This framework allows designers to replay a set of interactions with the Virtuoso Analog Design Environment without using a graphical user interface. Typically, the OCEAN scripts are automatically generated by one of the Virtuoso tools as a way of saving simulation configuration or analysis from the graphical environment. These scripts may then be tweaked and used as a part of a test plan.

The main advantage of an OCEAN-based solution stack is that it is easy to generate OCEAN from Cadence tools. That is, a designer who is familiar with simulation and analysis in the Virtuoso environment can quickly leverage OCEAN to repeat simulations and analyses without knowing much about OCEAN or anything about SKILL. However, this ease of use can be a double-edged sword. While it is possible to take a snapshot of a simulation and analysis setup and repeat it, the automatically generated code is somewhat messy and brittle. It may be difficult to use the same automatically generated OCEAN script on a different circuit or for different simulations. This trade-off is not surprising: by hiding the complexities of writing source code, it makes writing simulation and analysis scripts easier, but it also relieves one of producing code that is reusable, easy to communicate and understand, and maintainable, which are important aspects of the art of programming. OCEAN is a valuable part of the solution space because it enables automated regression testing, which is better than manual testing, even if it may be brittle or otherwise less than ideal. However, for our goal of promoting test reuse, an OCEAN-based solution stack would present many challenges.

Solution Stack	**Languages**	**Libraries**	**Simulators**
MATLAB, Python, SPICE	MATLAB, Python, SPICE	MATLAB	SPICE
OCEAN	SKILL	OCEAN built-in	Spectre
CircuitBook	Python, SPICE	CircuitBook	HSPICE, Spectre

Table 2.2: Comparison of Candidate Solution Stacks

2.4.2 Benefits

Using the CircuitBook framework to specify and run tests has several benefits. First, test descriptions tend to be shorter and easier to comprehend because CircuitBook provides a high level DSL (domain specific language) for test descriptions. Second, we try to separate the circuit specific[2] portions of test code from the application specifications. That is, the code that tells the system what stimulus to inject, what analysis to perform, how to measure the output, and how to analyze the results is separated from the code that configures the test framework for a particular instance of a circuit (e.g. port name mappings, process technology, etc.). This separation increases the potential for reuse because reusing entire tests is hard due to the need for some circuit-specific parts in every test; if we separate the circuit-specific parts of the test, then the remainder is much more reusable.

Third, by standardizing on a solution stack, productivity improves due to shared test collateral. This is already done to some extent at the workgroup level inside IC design houses for these reasons. However, that is just scratching the surface. Most of the sharing at the workgroup level constitutes test collateral for a particular design and its derivatives. There are many additional benefits to adopting a standard solution stack division-wide or company-wide. A standardized solution stack would naturally lead to common skills among engineers in various groups, which leads to better intra-company mobility, which in turn enables more efficient use of human capital. In addition, we believe that there is an opportunity to share parts of the test collateral (e.g., a particular way of measuring harmonic distortion) that would be applicable across different types of designs.

The second part of CircuitBook, the test repository, has additional benefits. A

[2]This is an application of the *separation of concerns* design principle in computer science.

global repository of test collateral will be a useful set of learning tools for people seeking to be better designers. Tests capture knowledge about the potential issues that can affect a design — tests are often created in response to surprise problems in a design. The CircuitBook repository, described in Section 4.11, helps users find circuits similar to the circuit they are trying to design. It also provides a way to discover test collateral that can serve as a basis for a new test. The challenge is in populating this repository. The repository becomes more valuable as it becomes bigger. Initial contributions are likely to come from universities and other groups without a direct profit motive. Once some basics exist, companies, EDA vendors, and industry consultants may be willing to improve and add on to the collection. MATLAB Central File Exchange is an example of this collaboration model. It contains over a thousand user-submitted Simulink models ranging from small components (e.g. models representing various modulation schemes) to complete systems (e.g., a three-phase induction motor drive).

In the next chapter, we will describe the CircuitBook framework and test repository in greater detail.

Chapter 3

Test Framework

The previous chapter provided an overview of analog circuit design and the ad-hoc manner in which testing is done today. This chapter introduces the CircuitBook test framework and associated repository as well as the supporting infrastructure. Our main goal in developing these tools is to make it easier to reuse tests by addressing the shortfalls of ad-hoc testing identified in Chapter 2. Users face two challenges when reusing tests: how to find test code to reuse and how to apply that code once it is found. CircuitBook addresses both issues. The CircuitBook repository helps users to find test code to reuse, and by defining a set of clean interfaces between test components, it makes repository code easier to apply to test new circuits.

In this chapter, we first review the circuit validation process and identify the natural interfaces in that process. These natural interfaces lead to a convenient decomposition into a set of tasks. Each of these tasks is in a different domain, so their natural forms are slightly different. We leverage the natural interfaces and the resulting decomposition in creating CircuitBook; these partitions allow us to create a domain-specific language (DSL) tuned for each task.

3.1 Test Components

Conceptually, each test consists of several parts: circuit representations, simulation directives, stimuli, measurement, and results analysis routines. This decomposition

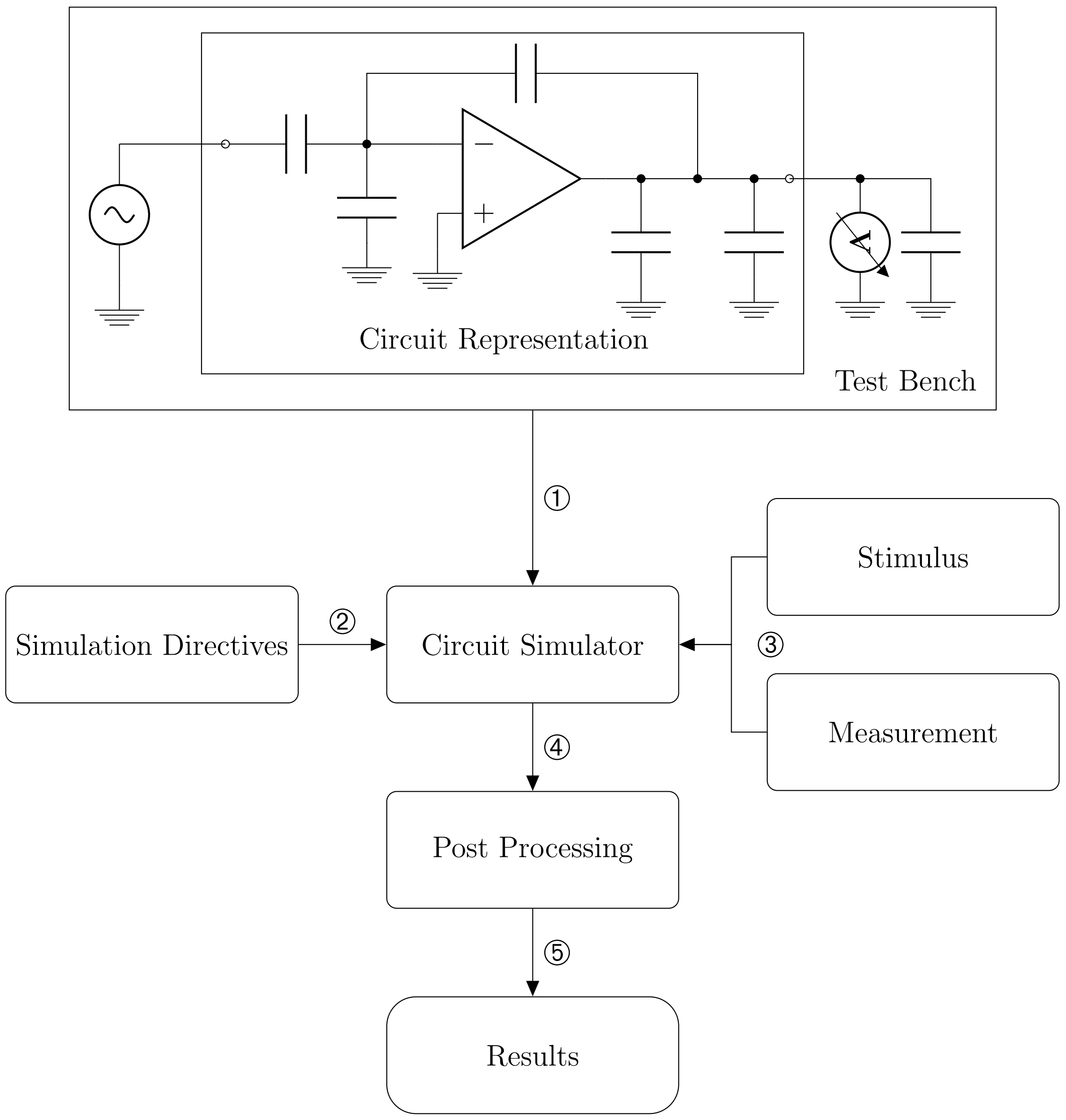


Figure 3.1: Decomposition of a Test — Circled numbers indicate the corresponding conceptual interface as described in Table 3.1.

is shown in Figure 3.1. We use this decomposition because each part has a different function, a different natural expression form, and a different set of reusable components. When constructing a DSL for writing tests, we want to provide different commands for each part to capture these differences. For the circuit representation, we need ways of describing circuit elements and the connections between them. This is traditionally expressed graphically in a schematic. Simulation directives are a combination of analysis type and simulator-specific options (e.g. the Gear integration method in HSPICE) that configure the simulator to perform the desired circuit analysis. The stimuli describe the inputs that will drive the circuit during the analysis. These input waveforms are generally specified by either their time or frequency waveforms. Simulator analysis of the circuit driven by the correct inputs results in the raw simulator outputs, which must be processed into the desired outputs.

The test process described so far, and depicted by Figure 3.1, shows a single test setup. For any real circuit, there are many of these test setups, often with only minor differences. For example, in testing an ADC, we want to drive the DUT with different sources for different measurements – a single sine wave for a code histogram, two-tone sine waves for measuring intermodulation distortion, and a ramp for evaluating non-linearity. The challenge in effective test reuse is to manage all these different test setups.

In an ad-hoc testing paradigm, as discussed in Section 2.3, the numerous test setups are often managed manually and written with no concept of distinct components. For example, consider how one would change the stimuli and measurements in an ad-hoc paradigm. One common approach is to have all of the necessary sources in a single file with all but one source commented out and likewise for the measurements. The relevant lines of code are often scattered throughout the file. Depending on the metric that needs to be measured, the appropriate combination of commands are uncommented. Another approach is to duplicate the entire set of test files and use a different set for each measurement. Neither of these approaches is ideal. For example, while each test component may only have a few variants, the combination of the components is exponential, resulting in a multitude of duplicate test sets or

Component	Ad-hoc Test	CircuitBook Test
Circuit ①	Circuit Netlist	Circuit Netlist with Metadata
Simulation Directives ②	Test Bench Netlist	`SimulationRun#setup`
Stimulus ③	Test Bench Netlist	`SimulationRun#start_run`
Measurement ③	Test Bench Netlist	`SimulationRun#start_run`
Post Processing ④	MATLAB Script	`SimulationRun#post_process`
Run Control	Shell Script	`TestRun` / `TestSequence`

Table 3.1: Location of Various Test Components in Ad-hoc Tests and in CircuitBook Tests — Circled numbers indicate the corresponding conceptual interface as labeled in Figure 3.1.

constant commenting and uncommenting. As can be seen in Table 3.1, each component in CircuitBook can be isolated, and changes need only be made in one portion of the code.

3.2 Conceptual Interfaces

In our test framework, we define a set of conceptual interfaces to provide structure to the test process and enforce the appropriate partitions. Each test component is specified using a DSL for describing that component type. This combination of DSL and interface forces users to decompose tests into pieces based on function and reusability.

Each of the numbered arrows in Figure 3.1 between components represents a conceptual interface. The interface labeled ① between the test bench and circuit simulator is the SPICE netlist. In the CircuitBook framework, we use `SimulationRun` scripts to model a single run of the underlying simulator. Simulation directives, labeled ②, are specified using a DSL in the `setup` function of `SimulationRun`. An example of this DSL is shown in Listing 3.1: the `set_simulator` and `analyze_dc` commands configure the type of simulator and the type of analysis. Similarly, the user describes stimulus and measurement via a DSL in `SimulationRun#start_run`. An example of this interface, labeled ③ in Figure 3.1, is shown in Listing 3.2: the

Listing 3.1: Example of `SimulationRun#setup` — This listing shows the setup of a typical test by defining the interface that the test leverages, the simulator to be used, the analysis mode, and what outputs to make available in post-processing.

```
class DiffampDCSimulation(SimulationRun):
    def setup(self):
        self.intent      = "Check␣saturation␣margin␣and␣power␣
           consumption␣of␣the␣Differential␣Amplifier"
        self.description = "Run␣a␣DC␣operating␣point␣simulation␣to␣
           obtain␣saturation␣margin␣and␣power␣consumption"

        # Set the interface we expect
        self.attach_to_interface('diffamp')

        # Load the HSpice simulator
        self.set_simulator(HspiceSimulator())

        # Run a DC analysis
        self.analyze_dc()

        # Specify which nodes and devices to save
        self.save_output(vsources=['vvdd'], nodes=['in_p', 'in_n', '
           out_p', 'out_n'])
```

`add_stimulus` and `add_stimulus_block` library functions instantiate a stimulus generator and a SPICE format stimulus block, respectively. For post processing, labeled ④, we provide objects, accessible as fields of `sim_results`, that encapsulate the simulation data; an example can be seen in Listing 3.3.

Using `SimulationRun` scripts to model a single simulator run has several key advantages. It separates the `SimulationRun` from environmental settings (e.g., process and corner parameters). This shared nothing architecture means that it is trivial to parallelize simulations. It also maximizes the reuse of these scripts because most of the information specific to a particular design are passed in, either explicitly (e.g., user-provided configuration in the `TestRun` object) or implicitly (e.g., site-wide configuration in the CircuitBook framework).

`TestRun` objects are a mechanism to run multiple simulations. Listing 3.4 shows a simple example that binds a technology process and runs a `SimulationRun` (i.e., the one described by Listings 3.1, 3.2, and 3.3) in each process corner.

Listing 3.2: Example of `SimulationRun#start_run` — In this example, we are defining the test bench for an amplifier. We first calculate the desired input common mode voltage from the supply voltage of the process corner being simulated. Next, we create power supplies and input stimuli using stimulus generators. Finally, we add a load specified as a SPICE snippet.

```
class DiffampDCSimulation(SimulationRun):
    def start_run(self):
        # Use the supply voltage from process corner
        supply_voltage = self.corner.supply()
        vincm = 0.6 * supply_voltage
        voutcm = 0.6 * supply_voltage
        itail_ideal = 1e-3
        rval = 2 * (supply_voltage - voutcm) / itail_ideal

        self.clear_stimulus()

        # Power Supply Stimulus
        self.add_stimulus(PowerSupplyStimulus(supplies = {'vdd' : supply_voltage}, grounds = ['gnd']))

        # Input Stimulus
        self.add_stimulus(DCVoltageSourceStimulus(pnode='in_p', nnode='gnd', value=vincm))
        self.add_stimulus(DCVoltageSourceStimulus(pnode='in_n', nnode='gnd', value=vincm))
        self.add_stimulus(DifferentialToSingleEndedConverter(inp='in_p', inn='in_n', outnode='indiff'))
        self.add_stimulus(DifferentialToSingleEndedConverter(inp='out_p', inn='out_n', outnode='outdiff'))
        self.add_stimulus(SingleEndedToDifferentialConverter(innode='indiff', outp='out2', outn='out2b'))

        # Add loading
        self.add_stimulus_block("""
            rldp vdd out_p {rval}
            rldn vdd out_n {rval}

            Cld  out_p 0 {cldval}
            Cldb out_n 0 {cldval}
        """.format(rval = rval, cldval = 100e-15));
```

Listing 3.3: Example of `SimulationRun#postprocess` — This post-processing routine shows how we extract the power of a voltage source from our result objects and emit that to the logging system.

```
class DiffampDCSimulation(SimulationRun):
    def postprocess(self):
        # Get the object that holds the simulation results
        dcres = self.sim_results.dc()

        # We saved the data for a voltage source named 'vvdd'.
        # dcres['vvdd'] returns a ComponentData object.  Voltage
        # sources have a current() and a power() method.  In this
        # case, we are using the power() to get the power as a
        # scalar.
        power = abs(dcres['vvdd'].power())

        # Create a new Epilog section
        with self.epl_section() as epl:
            # Print the power to stdout as well as the Epilog
            epl.println("Power: %.2fmW" % (power/1e-3))

        return {'power' : power}
```

Listing 3.4: Example of `TestRun` — This listing shows a typical example of `TestRun` where we want to run a particular `SimulationRun` against all the corners of a particular process. Note that this is very short; our framework tries to make these scripts as concise as possible since `TestRun` code is more specific and less reusable.

```
class DiffampDCTest(TestRun):
    def setup(self):
        # Load the Synopsys 90nm process model
        models = [SAED90ProcessModel(scale=0.050e-6)]

        # Instantiate the SimulationRun objects
        self.dct = self.construct('DiffampDCSimulation', models)

    def test(self):
        # Run the simulations across all corners
        for corner in self.act.corners():
            print "Corner: %s" % corner.name
            self.simulate(self.dct)
```

Now that we have provided an overview of the conceptual interfaces in our test framework, the next sections provide a more detailed description of these interfaces.

3.3 Circuit Representation

The circuit is the center of any test setup, and how it is represented drives the design of the rest of a test framework. In our system, circuits are stored as their netlist representations. A circuit object is essentially a parameterized SPICE `.subckt` that we treat as a black box. The main reason for this black box approach is predictability. We want it to be easy for designers to infer the operation of a particular circuit block, and this level of design is what they work with today. It also means that these circuit blocks can be used in other tool flows, since our representation of the circuit is its natural representation (i.e., how a circuit block interfaces with other circuit blocks). By describing circuits in this manner (i.e., only allowing parameterization through the standard SPICE convention for parameters), we implicitly assume that circuit topologies are fixed.

While we use the standard SPICE `.subckt` notation to create our black box representation, the black box abstraction we use is stronger than that of SPICE. We do not allow the user to look inside the box and probe nodes or measure devices directly, whereas SPICE allows this through its dot notation. This is an application of a design principle commonly used in object-oriented software development — the Law of Demeter. Having this isolation means that all values of interest (i.e., parameters or nodes) must be defined as ports. Resulting test code is cleaner because it cannot contain circuit specific code, since circuit elements are not visible except through ports. Circuit specific checks are kept with the circuit. For example, it is common to check whether the devices of a particular circuit are in the proper operating regions. Traditionally, this is done by probing the devices at the top level using dot notation to peek inside. In our approach, the circuit contains assertions that check whether devices are operating in the appropriate regions and exposes the result of those checks as a value through a port of the interface.

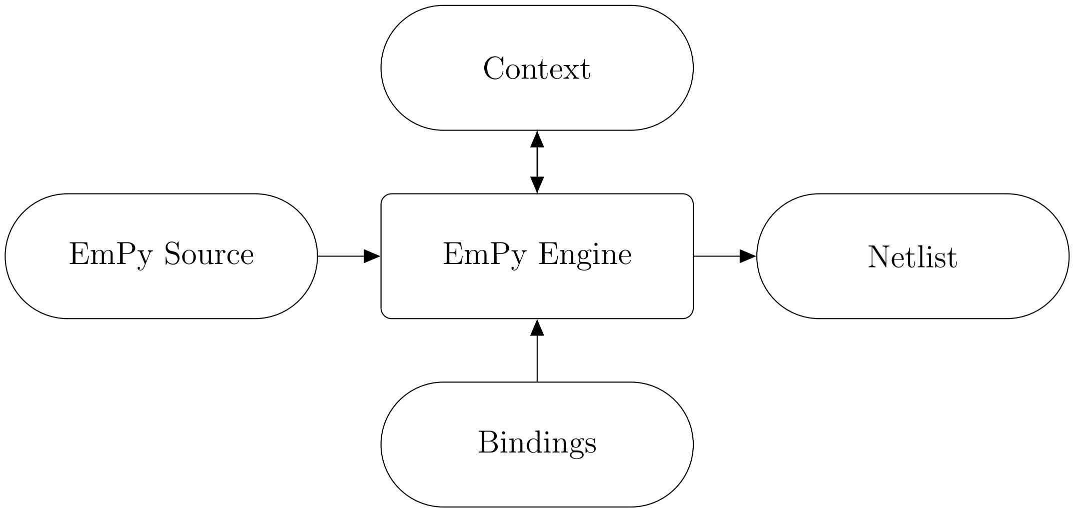


Figure 3.2: Elaboration Flow of the EmPy Template Engine

3.3.1 Problems with Template-Based Approaches

During the development of the CircuitBook test framework, we considered making the circuit description more flexible, by using EmPy[46], a general-purpose Python-based templating system, as the representation format for the circuit and test bench netlists. A general-purpose template system takes an input file that contains specially delineated regions and produces an output file where those regions are replaced with the appropriate text. When run, EmPy replaces occurrences of `@(var)` in the input with the value of the variable `var`. The collective set of these variables are the *bindings*. This flow is illustrated in Figure 3.2. For example, running the text,

```
Good @(part_of_day), @(name).
```

through EmPy with bindings of `name = "James", part_of_day = "morning"` yields

```
Good morning, James.
```

In addition to this basic string replacement, template engines usually also support conditionals and loops. For example, in EmPy, `@[if` ***`condition`***`]...@[end if]` is quietly removed from the source text if the *condition* does not evaluate true. Similarly, `@[for ...]...@[end for]` denotes a loop in EmPy.

Some template engines, including EmPy, provide evaluation support — the ability to embed a dynamic programming language such as Python inside the template. EmPy provides evaluation of Python expressions, that is, in addition to simple substitutions of variables, it will accept and evaluate a Python expression inside the `@(...)` construct.

These advanced template features are useful in two ways. First, they provide a more flexible way of handling circuit parameters in lieu of the SPICE `.subckt` parameter passing mechanism. For example, this allows Python expressions as values for device sizes in addition to SPICE expressions. Second, we can leverage these features in test bench construction. In the earlier version of CircuitBook, we used EmPy as a means of constructing the top-level test bench. We required that the user provide the circuit as an EmPy template ready for HSPICE simulation with some fixed placeholders (e.g., we required a `@(stimulus_includes)` line before the circuit and a `@(measurement_includes)` line after). To create the test bench, the framework expanded the user-provided template and replaced the placeholders with the generated output (i.e., `@(stimulus_includes)` is replaced with the output of the instantiated stimulus generator outputs).

Eventually we outgrew the benefits. In the process of making the framework easier to use and adding additional simulator support, we stopped requiring the user to provide the top level test bench template. In order to support both HSPICE and Spectre, we needed to abstract the analysis commands and simulator options, so we built a test bench generator as part of the system. This process of generating the test bench is described in Section 4.4.1. Automatic test bench generation was a clear win, since it made it simpler for the user and allowed us to be simulator agnostic while eliminating our dependence on the template engine.

At that point, the only use for templating was to customize circuit parameters. While that use seemed like a good idea in the beginning, it quickly proved otherwise. Designers ended up abusing this mechanism and writing SPICE netlists that contain lines that look like:

```
R1 input 0 @(load_res*1000)K
```

Note that the unit for `load_res` is actually `M`Ω since its value is multiplied by 1000 in Python and then scaled by the `K` unit in SPICE. The abuse did not end there. In our experience, designers sometimes overused the evaluation feature of the template engine and a large amount of test code ended up being written inside the template, which made the test and cell hard to reuse.

Dynamic evaluation is very powerful and when used incorrectly it is hard to reason about the behavior of a particular template. It takes discipline and best practices to avoid such improper use. Since designers are not programmers by trade, it is particularly unreasonable to expect them to follow best practices regarding the separation of templates and code. When isolation between templates and code is not maintained, subtle syntax errors can create errors that are difficult to debug.

After using templating, we determined that the additional flexibility made the system more difficult to learn and to use. On balance, it seemed that removing this templating capability made the methodology more robust by allowing only a single canonical way of performing a certain task. As a result, we use parameterized SPICE `.subckt` blocks as the basic unit of circuit representations. These blocks allow device parameters to be changed but constrains each parameter to a single token and thus avoids most of the issues discussed earlier with the use of templates.

Now that we have laid the foundation for our circuit representation, we will explain our method for categorizing circuits and how this representation is useful for archiving tests.

3.4 Interfaces

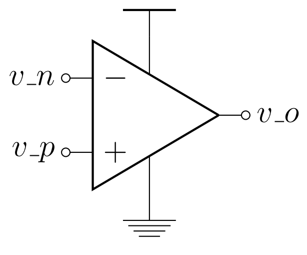
In the previous section, we presented our approach to representing circuits (i.e., as a black box). In combination with this black box approach we define a notion of circuit interfaces that allows the design of circuit-independent tests. An interface is a set of ports and associated metadata. Ports are either physical or informational. A physical port is a pin or a set of pins. An informational port can be the parameters of the circuit, assertions, or measurements.

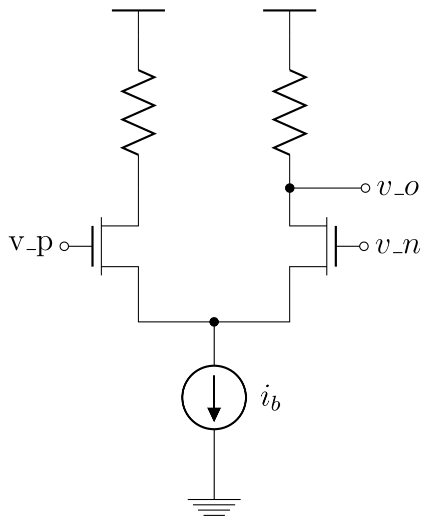
An interface separates a circuit from its tests by defining the structure of interaction between them, which allows us to write tests only using the abstraction of the circuit. A similar concept, symbols, is used in traditional schematic capture flows to manage complexity. Figure 3.3 shows this abstraction. There are a couple of differences. First, symbols typically only contain physical wires whereas our notion of interface also include information (e.g., error flags). Second, symbols are closely associated with a single circuit (i.e., a symbol is a just another view of the circuit) which makes the symbol subordinate to the circuit. Our definition of interfaces reverses these roles. We see the interface as the primary view. In our framework, tests do not see circuits at all; tests only interact with an interface. There are several implications of this stance:

- Test reuse across different circuit implementations exposing the same interface is trivial since the test does not know anything about the circuit and cannot access any of the circuit's internals during simulation.

- Tests can be lifted to higher level interfaces. For example, a test measuring the distortion of a particular amplifier interface can be separated into the theoretical calculations and the drivers and monitors for that interface. The first part is a test that works on all amplifiers and is defined at higher level interface. The second part is the mapping between the higher level interface and the lower level interface. Interface hierarchies are described in Section 3.8.

Interfaces provide these benefits by restricting what a designer can do in a test (i.e., they cannot probe the inside of the circuit). This makes writing a test a bit more complex. Designers must think about the information they need from a circuit and declare those as a part of the interface. This extra work is not wasted – it is very useful in creating a clean definition of the requirements of a test that can be used when testing actual silicon. The visibility restrictions avoid the unfortunate scenario of not being able to debug a chip because there are some nodes that are not visible. Any nodes needed for testing are captured by the interface definition created while constructing simulations.





(a) Interface (b) Circuit Implementation

Figure 3.3: Example of a Differential Amplifier Implementation Abstracted by an Interface

3.5 Stimulus and Loading

Designers usually create a test bench for testing circuits. These test benches contain input sources and output loads. Input sources drive the circuit, and output loads simulate the next stage and measure parameters of interest. While there are many ways to connect up these stimuli, loads, and measurement instruments, there are only a limited number of fundamental building blocks. This allows us to create abstracted representations that isolate the designer from the simulator, which then allows us to be simulator agnostic.

First, let us look at stimuli. It is easy to identify the basic building blocks; we need to look no further than the HSPICE Reference Manual[47]. We will use HSPICE as an example simulator for our discussion since its syntax is also compatible with its main commercial competitors (e.g., Spectre[7] and Eldo[48]). Circuit simulators generally provide three types of sources: independent sources, dependent sources, and miscellaneous sources. Independent sources are voltage sources or current sources that generate a time-series of voltage or current values based on a programmable expression. Table 3.2 shows the functions available in HSPICE. Dependent sources

Function	**Command**
Pulse	`PULSE`
Sinusoidal	`SIN`
Exponential	`EXP`
Piecewise linear	`PWL`
Single-frequency FM	`SFFM`
Single-frequency AM	`AM`

Table 3.2: HSPICE Independent Input Functions

Source	**Element**
Independent Voltage Source	`V`
Independent Current Source	`I`
Voltage Dependent Current Source	`G`
Current Dependent Current Source	`F`
Voltage Dependent Voltage Source	`E`
Current Dependent Voltage Source	`H`
Digital Files and Mixed Mode	`U`

Table 3.3: HSPICE Stimulus Elements

are voltage or current sources that are controlled by another voltage or current measurement. Other sources typically include sources which load values from an external source, such as a waveform from a digital simulator. These different types of sources are summarized in Table 3.3.

The CircuitBook test framework provides a set of generators that abstract these basic sources. For example, `DCVoltageSourceStimulus` generates an independent voltage source with a DC value and `PWLVoltageSourceStimulus` generates that same source with a piecewise linear input function. We try to explicitly define each combination of stimulus element type and input function as a separate class. `DCVoltageSourceStimulus` and `PWLVoltageSourceStimulus` both map to `V` elements in HSPICE notation. These explicit definitions capture design intent, and we leverage that intent to produce metadata about each test, which are then used in the rest of the tools. In particular, this is very important in helping us organize tests in our repository. It may seem burdensome to produce so many different generators, Listing 3.5: Framework Source Code for `PWLVoltageSourceStimulus` — This stimulus generator leverages `AbstractSourceStimulus` heavily; it sets the appropriate options and passes control to the parent. This is a common pattern that helps make stimulus generators more reliable by avoiding duplicated code.

```python
class PWLVoltageSourceStimulus(AbstractSourceStimulus):
    """ PWL voltage source to be instantiated between *pnode* and *nnode
       *.
        Parameters are:
        - wave : an array of time-value pairs given in a string e.g.,
            "[1n 0 1.1n 1]"
    """

    def __init__(self, pnode, nnode, wave, name = None):
        super(PWLVoltageSourceStimulus, self).__init__(pnode, nnode, 'V'
            , name = name, wave = wave, type='pwl')
```

but it is not. There are only a few commonly used input functions (i.e., a subset of those listed in Table 3.2) and two independent source types (i.e., voltage and current), so the number of combinations is small. Since each generator is only a couple of lines of code, it is easy to write and maintain. See Listing 3.5 for a example of such a generator declaration; that code simply defines `PWLVoltageSourceStimulus` as a subclass of `AbstractSourceStimulus` and handles some parameter mapping. The heavy lifting is performed in `AbstractSourceStimulus`, which is shared by many stimulus generators.

Independent sources are often used to represent inputs in the real world, while dependent sources are often used as glue in simulations. For example, dependent sources are used to construct a single-ended to differential converter to make it easier to describe the stimulus. Instead of writing the functions for a pair of differential inputs, it is common to use a pair of independent voltage sources to represent the differential and common mode voltages and then convert these into different inputs using some dependent sources. Again, we provide abstractions for these types of blocks (e.g., `DifferentialToSingleEndedConverter` and `SingleEndedToDifferentialConverter`), in addition to abstractions for basic dependent sources, to allow the users to declare their intent.

We built these generators as users needed them, so there are probably additional generators that may be useful. Instead of trying to write every stimulus generator, we focused on building the infrastructure to facilitate user-built generators. Our stimulus generation framework, described further in Section 4.4, allows users to easily make larger generator blocks by composing smaller generators and provides many helper functions that can be used in writing stimulus generators (i.e., there is a DSL for writing generators). These stimulus generators decouple the declaration (i.e., the what) from the implementation (i.e., the how) and allow us to construct tests that can work on many different back-ends: simulators (e.g., Verilog, SPICE), test equipment, and automatic testers.

We use these same concepts in handling loads. Loads are simply compositions of circuit elements and can be built on the same generator DSL that is used to make stimulus generators. While we currently do not provide any predefined load generators, the process of creating them is straightforward. We have observed users construct their own load generators to share across their tests.

3.6 Measurement and Post Processing

The basic measurement commands provided by circuit simulators are usually very simple. In HSPICE, the `.PROBE` command saves, and optionally plots, various simulation variables. However, the measurement apparatus created by users are often very complex. This disconnect creates problems in reuse. When users are forced to build complex systems from simple primitives without good abstractions and interfaces, the result is chaos: measurements for different performance metrics are intermingled, making it hard to pull out a particular measurement to reuse. Furthermore, there is not a well-defined interface between measurement and post processing in ad-hoc testing. One designer calculates duty cycle via a SPICE expression in a `.PROBE` statement, whereas another designer extracts the raw waveform and processes it in MATLAB.

To avoid these problems, we define a very clear interface between measurement and post processing. Users specify the data they are interested in (e.g., via a call to `save_node`, `save_vsource`, `save_isource`, or `save_transistor`), and the framework automatically provides those results as objects in the post processing routine. All data traditionally available in a SPICE simulation are available through the appropriate `save` call, including node voltages, device currents and parameters, and voltages and currents of sources. However, any internal nodes or values used must be declared and visible on the interface as we mentioned in Section 3.4.[1] When stimulus generators are used, the results are automatically associated with the appropriate objects. For example, a `PowerSupplyStimulus` will automatically load the relevant voltages and currents and provide that data in a special results object for power supplies. These special objects keep the meaning of the data with the data. As a result, specialized calculation functions can be defined on these objects (e.g., a function to calculate settling time) and these objects can be also passed to Epilog, our logging system described in Section 4.7.

This process is simulator agnostic; our result objects abstract away the differences in data formats between various simulation engines and provide a consistent interface to the user. Essentially, the user focuses on writing post processing routines without having to worry about how to get the data. The only measurement tasks the user has to do is to indicate the nodes of interest. This helps us in two ways: it ensures optimal performance by minimizing data transfer, and it serves as another source of metadata. Section 4.6 describes the process in detail and discusses the programming interface.

3.7 Types of Tests

Before we dive into how tests are organized, we need to separate tests into two types: performance tests and assertions.

Performance tests represent design specifications for the circuit. For an ADC, we would like to measure, at a minimum, the signal-to-noise ratio, total harmonic distortion, differential non-linearity, integral non-linearity, and noise power ratio[49,

[1]While this is generally true, we have a debug mode where this constraint is relaxed so as to make it easier for users to experiment. In that debug mode, commands such as `save_transistor` will accept a dotted path to the device (e.g., `amplifier.M5`).

50, 51, 52, 53]. Since these performance tests are relevant to any ADC, they would be attached to the ADC class and thus available to all ADCs. Depending on the application of the circuit, additional performance metrics of interest may apply. For a flash ADC in a radar application, we might also want to measure the transient response to a square wave input[54]. Such a test would be attached to the flash ADC class.

The other type of test is assertions. Assertions are tests that verify whether constraints are met. Like assertions in software, these are pass/fail checks inside a test that guarantee assumptions are met. For example, a certain port may be a LVDS (low-voltage differential signaling) input, and the voltages on that port should always fall within the range provided by the LVDS standard (i.e., ANSI/TIA/EIA-644-A). Failure to meet the standard may lead to performance tests passing but the circuit failing in production. In general, the fidelity of performance tests is compromised when assertions fail. Another example of a common assertion is a check that a particular device stays in some operating region. A failure in such an assertion may represent a marginal design, and the degradation is likely visible in the results of the performance tests. Assertions are useful here because they make it easy to figure out the root cause.

3.8 Organizing Circuits and Tests

Given our circuit representation, a key reuse problem is how to find tests to reuse. While there might be a circuit similar to yours, you have no idea what the other designer called it. To address this issue, we leverage the notion that circuits belong to a tree-based classification system. This categorization is important for two main reasons. First, it allows the automatic adaptation and reuse of tests. Second, this classification is a natural way to browse the repository for tests and for designers to discover circuits or tests that they may wish to reuse.

We categorize circuits into tree structures based on the circuit function. The root of each tree represents a class of circuits and the leaf nodes are individual circuits. For example, the path from a particular implementation of a flash ADC to the root of its

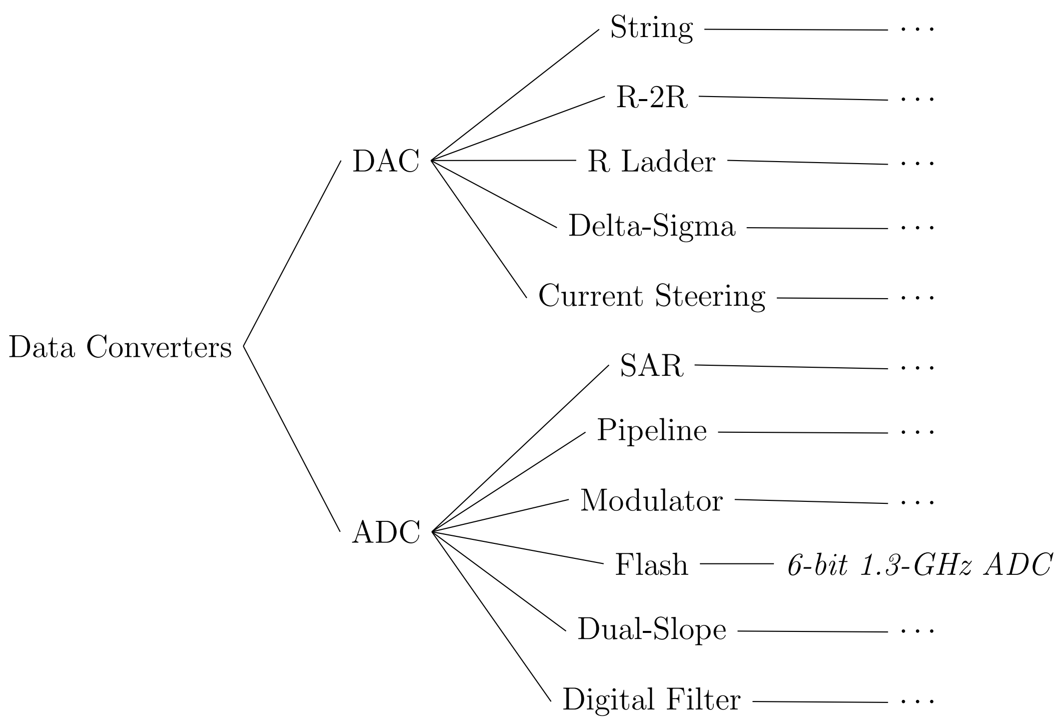


Figure 3.4: Example of a Data Converter Hierarchy

tree may look like this: Flash ADC to ADC to Data Converters. This is illustrated in Figure 3.4. While it is possible for a single circuit to map to several leaf nodes, that is not the common case.

In order for the tree structure to be intuitive, a designer must be able to quickly identify the properties of a certain class of circuits. One can imagine this happening in one of a few ways. This can be done by looking at that circuit class and its description; if needed, the designer can also explicitly examine the tests attached to that particular class. The designer can also gain insight into the properties of a circuit class by looking at the circuits that belong to that class. The end result is that a circuit class in an intuitive hierarchical categorization system must have a defining set of properties at each level. These properties then can be checked by tests at this level which allow us to "lift" some tests to higher levels in the hierarchy; this is discussed next.

3.8.1 Class Hierarchy

Clearly, any property that is relevant for a circuit class must be relevant for all subclasses as well as any circuits belonging to that class or a subclass of that class. Similarly, any test defined on a circuit class must also be relevant for any circuits belonging to that class or one of its subclasses. This relationship logically leads to test reuse.

Consider an inner node in the classification tree (e.g., the data converter hierarchy shown in Figure 3.4). Such a node represents a certain class of circuits. All of the tests attached to this node are related to a property relevant to the circuit class and should be evaluated on all circuits under this node (i.e., leaf nodes that are descendants of this node). In addition, each of these tests should be executable on any of the circuits. That is, these tests should be written in a way that allows them to be reused across different circuits. The key to doing so is defining the boundary of interaction between the test and any devices under test. This boundary is the interface described in Section 3.4. Tests are attached to circuit classes through these interfaces. Table 3.4 shows an example of the classes in such a classification.

3.8.2 Adapting Tests to Circuits

By using formal interfaces, we can automatically generate the plumbing required to make a test work with any circuit that implements an interface lower in the tree. A test defined on the interface of the ADC class can be run with a circuit that supports an interface attached to a subclass of the ADC class (e.g., flash ADC). We use the circuit metadata, described in Section 4.1.1, that describe the mapping between the circuit and the interfaces it directly supports. This allows us to construct a circuit with the formal interface of the test. Interfaces have similar metadata mapping them to their parent interfaces. That is, a flash ADC interface must provide a mapping to the interface for its parent class (i.e., ADC).

Concatenating these connections mean that test collateral can be lifted in the hierarchy. When a test is initially developed, it is written at an interface associated with the circuit being tested. Reuse is limited because a different circuit likely has

Class	**Subclass**
Amplifiers and Comparators	
Clock and Timing	Clock and Data Recovery/Retiming
	Clock Generation and Distribution
	PLL Synthesizers/VCO
Data Converters	ADC
	DAC
	Switches and Multiplexers
Interface	Level Translators
	LVDS
Power Management	Current Sources
	Linear Regulators
	Switching Regulators
Reference	Current Reference
	Series Voltage Reference
	Shunt Voltage Reference
RF/IF	Downconverting Mixers
	I/Q Demodulators
	I/Q Modulators
	Upconverting Mixers

Table 3.4: Examples of Circuit Classes

a different interface. We lift the test by re-writing it so that it conforms to a higher level interface. This makes the test reusable on a wider range of circuits. Lifting need not be performed on an entire test; it can be done on a part of a test, since tests can be written as compositions of smaller tests. Once lifted, a test is run on all the circuits below it. This means that a newly discovered bug, captured in a test, is now screened for whenever any child circuit is tested.

Interfaces allow us to connect circuits with tests elsewhere in the hierarchy by providing metadata that allows the system to construct the necessary bridges. Each interface definition contains metadata that define mappings to other higher level interfaces. Often, this is simply a list of port mappings, but sometimes a little additional circuitry is needed. For example, a circuit may need a converter that converts a common-mode single-ended input into a differential input or something to set the correct common-mode voltage of the differential output.

3.8.3 Indexing / Searching

The circuit hierarchy also allows our repository to provide a couple of different mechanisms for locating the desired test: hierarchical browsing, metadata-based browsing, and tag-based browsing. Hierarchical browsing is quite straightforward – circuits and tests are defined against interfaces, which means that tests can be browsed by browsing the hierarchy of interfaces. Figure 3.5 shows the interface browser in the CircuitBook repository. The circuit class hierarchy is visible on the left and the details of the selected circuit class is presented on the right. As we discussed in the previous subsection, a circuit conforming to interface `X` can be tested with any test written against the interface `X` or one of its parents. Therefore, given that a user has a particular class of circuit to test, all compatible tests can be quickly retrieved.

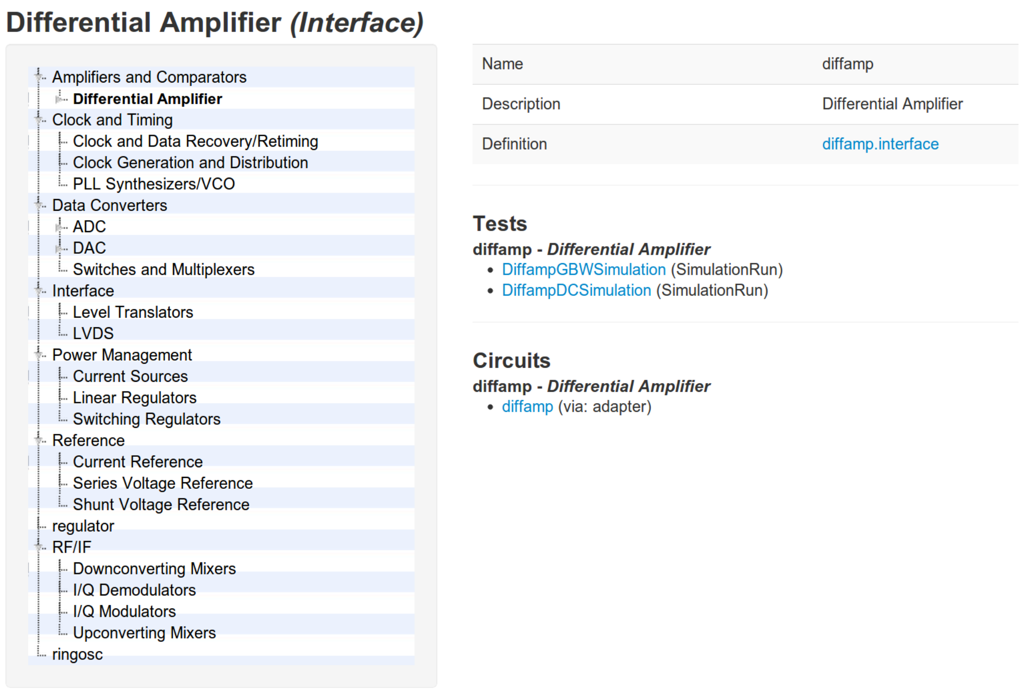
Figure 3.6 shows the different `SimulationRun` objects and their associated interfaces called by a particular `TestRun`. The graph shows `TestRun` objects as rectangles, `SimulationRun` objects as rounded rectangles, and interfaces as ovals. The top level test is a `TestRun` called `DiffAmpTest` which calls four `SimulationRun` objects which are defined against two different interfaces, `diffamp` and `diffamp_bias`. From this view, we can quickly determine that this `DiffAmpTest` can be run on any circuit that can support the `diffamp_bias` and `diffamp` interfaces.

Earlier in this chapter, we discussed the various test components and how they are supported by our test framework through the use of DSLs and that those DSLs are designed for ease of metadata extraction. When tests are loaded into the repository, we parse the test to generate metadata and use that metadata as an index to tests. Figure 3.7 shows some of this metadata for a `SimulationRun` object. Any analyses used or stimulus generators invoked are identified and listed. Clicking on a listing queries the system for other tests with the same property. Low-level metadata (e.g., the data related to which library functions were used) are analyzed through heuristic rules to infer higher level metadata (e.g., tags).

Tag-based browsing relies on the notion that each test is tagged with a variety of metadata tags derived from test class. These tags are automatically generated when certain framework features are used (i.e., by inspection of the abstract syntax tree of the test code after it is loaded into Python). The user is able to manually add tags by specifying intent, but the system does not rely on the user's diligence for operation. There are several methods to browse by tags. The user can search for all tests containing certain tags (e.g., all tests that work in the frequency domain) or look for similar tests to a particular seed test (e.g., the user has a test for a particular circuit class and wants to find a test that measures similar specifications for a different class). From Figure 3.7, we see that this particular test has three tags attached to it: *Technique: Frequency Domain*, *Measurement: Frequency Domain*, and *Application: Amplifier*.

In the machine learning sense, the problem of finding similar tests is a recommendation problem – that is, we wish to predict which tests would interest a user when they are looking to pick a test to reuse[55]. Our tag-based browsing is a straightforward implementation of a recommendation system. First, we build a feature vector for each test. In our tag-based system, the feature vector is simply a boolean vector with all the possible tags, where each position indicates whether the test is tagged with that tag.

When the user performs a search, the search query is first mapped to a feature



Differential Amplifier *(Interface)*

- Amplifiers and Comparators
  - **Differential Amplifier**
- Clock and Timing
  - Clock and Data Recovery/Retiming
  - Clock Generation and Distribution
  - PLL Synthesizers/VCO
- Data Converters
  - ADC
  - DAC
  - Switches and Multiplexers
- Interface
  - Level Translators
  - LVDS
- Power Management
  - Current Sources
  - Linear Regulators
  - Switching Regulators
- Reference
  - Current Reference
  - Series Voltage Reference
  - Shunt Voltage Reference
- regulator
- RF/IF
  - Downconverting Mixers
  - I/Q Demodulators
  - I/Q Modulators
  - Upconverting Mixers
- ringosc

Name	diffamp
Description	Differential Amplifier
Definition	diffamp.interface

Tests

diffamp - *Differential Amplifier*
- DiffampGBWSimulation (SimulationRun)
- DiffampDCSimulation (SimulationRun)

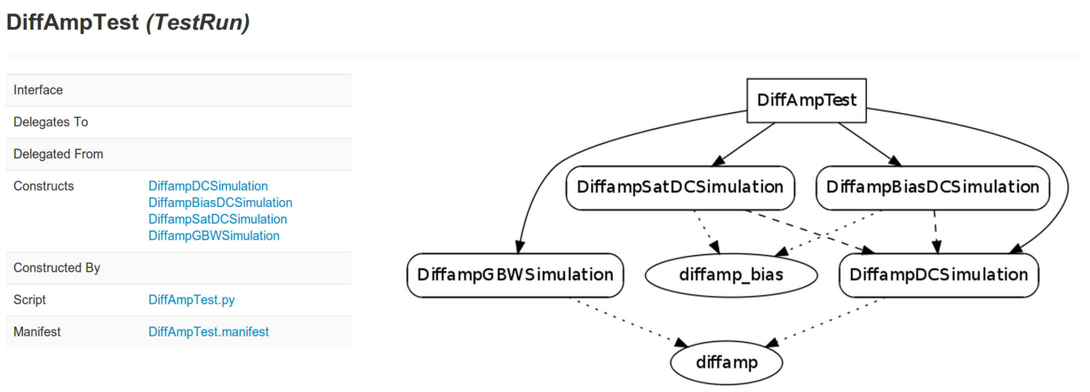
Circuits

diffamp - *Differential Amplifier*
- diffamp (via: adapter)

Figure 3.5: Screenshot Showing Circuit Interface Browser

vector and that query feature vector is compared to all the tests in the system using a similarity metric to find the most closely related tests. Search queries can be text strings (i.e., keywords or phrases) or a test (i.e., find other tests similar to a selected test). The latter case is straightforward; the feature vector of the selected test is the query feature vector. In the text query case, we first perform a full-text search on the name and description fields of tests to find some candidate tests and then use a weighted average of those candidates' feature vectors as the query feature vector.

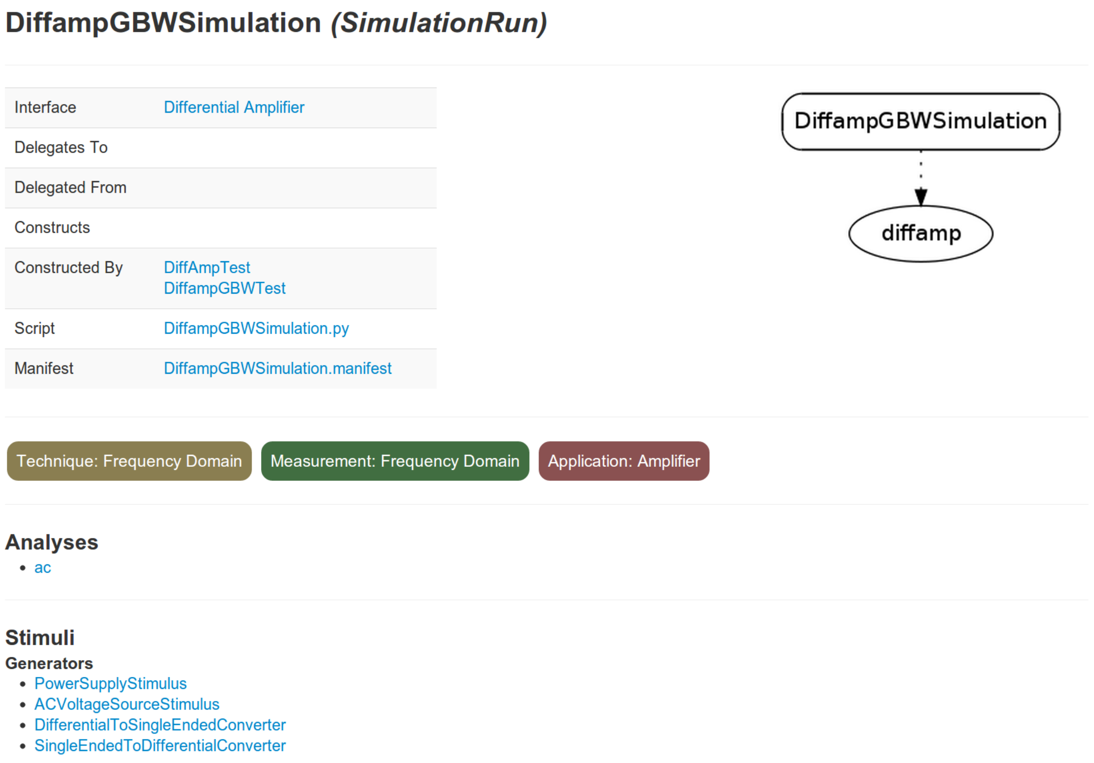
Once we have the query feature vector, we want to find closest tests in vector space. A simple implementation would simply calculate the distance between the query feature vector and the feature vector of each test, which works well for small repository sizes. For large number of tests, there are many probability approaches which can complete much faster (e.g., a locality sensitive hashing technique such as Simhash[56]). However, we do not currently have the number of tests to necessitate such techniques.



DiffAmpTest *(TestRun)*

Interface	
Delegates To	
Delegated From	
Constructs	DiffampDCSimulation DiffampBiasDCSimulation DiffampSatDCSimulation DiffampGBWSimulation
Constructed By	
Script	DiffAmpTest.py
Manifest	DiffAmpTest.manifest

Figure 3.6: Screenshot Showing Test Dependencies



DiffampGBWSimulation *(SimulationRun)*

Interface	Differential Amplifier
Delegates To	
Delegated From	
Constructs	
Constructed By	DiffAmpTest DiffampGBWTest
Script	DiffampGBWSimulation.py
Manifest	DiffampGBWSimulation.manifest

Technique: Frequency Domain | Measurement: Frequency Domain | Application: Amplifier

Analyses
- ac

Stimuli
Generators
- PowerSupplyStimulus
- ACVoltageSourceStimulus
- DifferentialToSingleEndedConverter
- SingleEndedToDifferentialConverter

Figure 3.7: Screenshot Showing Test with Inferred Metadata

3.9 Summary

This chapter focused on the use of formally defined interfaces to create reusable tests. We use the natural boundaries in a test, shown in Figure 3.1, to define interfaces between test components in our framework. By defining formal interfaces that separate tests and circuits, it is easy to reuse tests, since they are written against an interface instead of directly against a circuit. Interfaces are classified in a hierarchy and we can map between such interfaces, which then allows us to reuse tests at different levels of the interface hierarchy. In addition, by leveraging clean interfaces, we foster separation of concerns resulting in more modular code and produce metadata for use in test discovery.

Chapter 4

Framework Components

In the previous chapter, we explored the rationale behind our decomposition of the test collateral. Now, we will examine how the CircuitBook test framework operates. We will start with an overview, and then we will take an in-depth look at the various components.

As Chapter 3 described, a basic CircuitBook test consists of several parts: a circuit representation, an interface definition, a `SimulationRun` object, and a `TestRun` object.

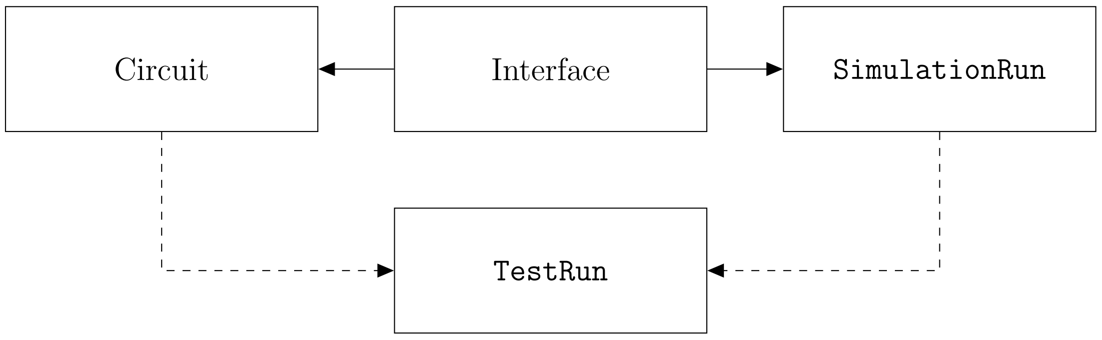


Figure 4.1: Test Partitioning

4.1 Circuit Representation

Circuits are represented in the CircuitBook framework as a SPICE subcircuit described by a netlist combined with some serialized metadata (i.e., metadata formatted in a way such that it is machine readable). An example of a complete circuit file is presented in Listing 4.1. The SPICE `.subckt` format is interoperable with almost all circuit simulators. Using this format helps to avoid any possibility of circuit mangling as previously discussed in Section 3.3.

We store a variety of metadata along with the circuit netlist; the metadata is serialized and prepended to the netlist. This metadata contains details about the circuit, such as name, parameters with default values, and ports, as well as the interfaces with which this circuit can interact.

4.1.1 Metadata

At the top level, the metadata is essentially a single dictionary with key-value pairs. These straightforward fields define the device-under-test (DUT). The `name`, `params`, and `ports` keys define the black box represented by the subcircuit in the netlist. These fields should match the `.subckt` defined in the netlist. The `name` field takes a string that is the name of the block. The `params` field is an array of dictionaries that define the parameters of the subcircuit; each such definition has a `name` field with a string for the name of the parameter and `default` field with a string for the default value of the parameter. Similarly, the `ports` key points to an array of dictionaries that define each port. Each dictionary has a single `name` field with a string for the name.

The metadata also contain information on how this circuit conforms to different interfaces, as explained in Section 3.4. Here, we are describing the details of how such metadata is specified in our framework. The `implements` key refers to an array of implementations of interfaces. Each implementation is defined by a dictionary. In this dictionary, there is a `name` field that specifies the interface that this implementation targets, a `via` field that specifies the mechanism to perform the actual implementation, and an `implementation` field that provides the parameters for the

Listing 4.1: Circuit File Example for a Differential Amplifier with Current Bias

```
---
name: diffamp
params:
- {default: ’200’, name: wnt}
- {default: ’200’, name: wns}
- {default: ’120’, name: wn0}
- {default: ’2.0’, name: ln0}
- {default: ’2.0’, name: lns}
ports:
- {name: vdd}
- {name: in}
- {name: inb}
- {name: out}
- {name: outb}
- {name: bias}
implements:
    -
        name: diffamp_bias_interface
        via: port_map
        implementation:
            ports:
                vdd   : vdd
                in    : in_p
                inb   : in_n
                out   : out_p
                outb  : out_n
                bias  : ibias
            params:
                wnt   : w
    -
        name: diffamp_interface
        via: adapter
        implementation: |
            xadaptee vdd in_p in_n out_p out_n bias diffamp wnt=w
            ibias vdd bias 1e-3
---
.subckt diffamp vdd in inb out outb bias wnt=200 wns=200 wn0=120 ln0=2.0
    lns=2.0

mn0 outb in  tail 0 NMOS w=’wn0’ l=’ln0’ m=’1’
mn1 out  inb tail 0 NMOS w=’wn0’ l=’ln0’ m=’1’

mnt tail bias 0 0 NMOS w=’wnt’ l=’lns’ m=’1’
mns bias bias 0 0 NMOS w=’wns’ l=’lns’ m=’1’

.ends
```

implementation. We can think of the `via` field as the name of a constructor to call and the `implementation` field as the data passed to that constructor. Currently, we have two such constructors implemented. There is a port mapping constructor (`via: port_map`) and an adapter constructor (`via: adapter`).

The port mapping constructor simply renames each of the ports and parameters of the block to match those of the interface. The `implementation` section for this constructor take a `ports` field with a dictionary of port mappings and a `params` field with a dictionary of parameter mappings. The keys in these mapping dictionaries correspond to the names used in this circuit definition, and the values correspond to the names that the ports and parameters should have to conform to the interface. In the example provided in Listing 4.1, the mapping for the `diffamp_bias` interface shows that the `out` and `outb` ports of this circuit are mapped to the `out_p` and `out_n` ports in the interface definition. When a test is executed, the framework will use this information to automatically create the instantiation of the DUT.

The adapter constructor allows the manual specification of a DUT instantiation. Our circuit example shown in Listing 4.1 is a differential bias amplifier with a current bias input. Some tests may not care about this bias input and may simply want to run the DUT with a fixed bias. In such a case, we can use the `implementation` field of the adapter constructor to provide an augmented circuit that includes additional test bench components. It first creates an instantiation of the DUT with the subcircuit called `xadaptee`. In our current example, we perform a simple port mapping and add an ideal current source to bias the circuit in order for it to conform to the `diffamp` interface.

The interfaces used in this example are provided Listings 4.2 and 4.3.

4.1.2 Metadata as an Abstraction

We have chosen to store all of the critical circuit information needed by the tools in the metadata; our entire suite of tools uses this information to interact with the circuit. Note that some of this information is also available in the circuit netlist, so we duplicate it in the metadata. While it may seem that duplicating this information in the metadata is bad, there are several advantages to this approach. We are essentially adding a level of abstraction between the framework and the raw netlist. This reduces the coupling of our tools, which makes it more flexible.

Suppose we did not have this information in the metadata. In that case, each tool that needs to interact with the circuit file must parse and extract the information needed from the circuit representation. If we did not have a reusable component to provide this functionality, we would have duplicate parsers in the framework, which would cause maintenance issues. To avoid this, we would extract the functionality into a separate module to enable reuse of this common function. This solves the code reuse problem, but now we are running this parser repeatedly in every tool unnecessarily; we only need to extract the information once. In short, storing the information in the metadata can be understood as a way of caching the information.

Efficiency aside, there is an even more important reason to store the circuit connection information in the metadata: to enable inspection and verification. That is, by having this information in the metadata and using an appropriate human-readable serialization method for the metadata (explained below), users of the framework are able to make sure that the connections are being interpreted correctly. This removes a key source of uncertainty. It frees us from worrying about the correctness of parsing the netlist. This separation of the tools from the raw netlist also makes it easier to use different simulation back-ends. That is, the tools are more flexible and applicable in a wider set of use cases by allowing us to use alternative circuit representations (e.g., Verilog model or actual silicon connected via test equipment).

4.1.3 Metadata Encoding

We combine the netlist and metadata together in one file to ensure that the two are never out-of-sync. While the synchronization of metadata and netlist can be maintained using a configuration management tool such as IC Manage[57] or a source code management tool such as Git, SVN, CVS, or Perforce, a simple misconfiguration can cause the files to become not synchronized. If we keep the metadata and netlist in one file, then the user does not need to worry about this potential problem.

Name	Text-based	Schema	References
XML	Yes	Yes	Yes
YAML	Yes	No[1]	Yes
JSON	Yes	No[1]	No

Table 4.1: Comparison of Data Serialization Formats

Since our circuit is stored as a SPICE netlist, a text-based format, the metadata must also be text if it is to be stored in the same file. We chose to use YAML[58] as the data serialization method for our metadata.

YAML is one of three common text-based data serialization formats; the other two are XML and JSON. All three formats are widely supported: there are libraries for serializing and deserializing these formats in most of the popular programming languages. More importantly, these formats are supported with syntax highlighting and formatting by many text editors (e.g., Vim, Emacs, UltraEdit, Sublime Text).

We chose YAML for our application based on ease of use. Generally, users may want to inspect and modify the metadata for a circuit design. In the case where circuit blocks are being reused, most users may want to read and understand the metadata, while only a small portion of users need to edit the metadata. Hence, ease of use, and particularly ease of reading the metadata, is an important concern. YAML and JSON are easier to read and write by hand, whereas XML is more verbose and does not map directly into the data structures commonly used in software (i.e., arrays and dictionaries). The choice between YAML and JSON is easy because JSON is a subset of YAML 1.2. The main difference is that YAML allows for references.

Note that, while our framework uses YAML libraries to read the metadata, we do not currently use any YAML specific features, so technically the metadata is also JSON compatible.

Listing 4.2: Interface Example for a Differential Amplifier

```
format: v1
version: 1
name: diffamp
ports:
    - name: vdd
    - name: gnd
    - name: in_p
    - name: in_n
    - name: out_p
    - name: out_n
params:
    - name: w
      default: 432
```

Listing 4.3: Interface Example for a Differential Amplifier with Current Bias

```
format: v1
version: 1
name: diffamp_bias
ports:
    - name: vdd
    - name: gnd
    - name: in_p
    - name: in_n
    - name: out_p
    - name: out_n
    - name: ibias
params:
    - name: w
      default: 432
```

4.2 Interface

In Section 3.4, we presented the idea of formal interfaces that define the connection (i.e., ports and parameters) between tests and circuits which increase the reusability of tests by decoupling tests from circuits. Now, we will explain how these interfaces are described in the CircuitBook test framework.

Interfaces are specified in a YAML-encoded `.interface` file with a single dictionary. This dictionary is similar to the dictionary used for circuit metadata described in Section 4.1.1. The `name`, `ports`, and `params` fields have the same meanings as in the circuit metadata with the exception that the `name` field now refers to the name of the interface instead of the name of the circuit. The top-level dictionary contains two additional fields: a `format` field that specifies the format of the interface definition for backward compatibility purposes and a `version` field that specifies the version of this interface definition. The `format` field is currently forced to be `v1` but may change in the future. This field allows future framework versions to understand older interface definitions. The `version` field is for handling interface changes, which allows for future framework versions to be backward compatible. This is important, since interfaces are globally shared across all circuits in a repository.

Listings 4.2 and 4.3 show two different, albeit very similar, examples of interface definitions.

4.3 `SimulationRun`

Having defined the circuits, and the narrow interface that tests can access, we now describe `SimulationRun`, the place where the specifics of a test are encoded. The `SimulationRun` script can be viewed as a collection of code snippets, which the user creates to control the flow of the simulation. These snippets of Python code supply the core of tests in the CircuitBook framework. One or more such scripts can be combined with `TestRun` scripts (see Section 4.8) to perform complex system-level simulations.

[1] Although there is no schema support for YAML or JSON in the standard libraries, there are some third-party attempts at creating tools to validate schema. The most popular of these are `Kwalify` and `Rx`. While both packages seem functional, the packages are not widely used or maintained.

There are two required snippets that perform setup and post-processing, called `setup` and `postprocess` respectively. In addition, there are some optional snippets (e.g., `start_run` and `end_run`) that are executed in various parts of execution flow.

In this section, we will first provide a simple mental model that is sufficient for most users of the system. The technical details of how `SimulationRun` scripts work is complex and will be explored in depth later. These details are primarily used by developers who want to extend or modify the system internals.

4.3.1 Mental Model

Since `SimulationRun` is automatically called by the framework, we will focus this discussion on what gets executed, rather than how to execute it. First, a `SimulationRun` instance is constructed in Python using the standard object construction mechanics. After that, the `setup` snippet is executed to provide the created object with its configuration information. Once that is done, the object is ready to execute simulations. For each simulation run performed, the `start_run` and `end_run` snippets are called before and after the actual underlying simulator is run, respectively. Finally, `postprocess` is called with objects representing the output of the simulation. This execution flow is presented in Figure 4.2.

`setup` is used for framework configuration that do not change between executions of the same simulation, whereas `start_run` is used for setting configuration for each run. A typical `setup` snippet specifies the type of simulator to run (e.g., HSPICE or Spectre), the type of analysis desired (e.g., DC, AC, transient, PSS), the outputs that should be saved, and the interface used, along with any user annotations about intent. `start_run` can be used to construct, programmatically, the test bench (i.e., it specifies the stimulus and loading for the DUT). `end_run` is usually not needed, but the callback is provided by the framework in case the user has required clean-up. For example, such a mechanism is helpful if the user needs to acquire and release simulator licenses manually; in that case, the license can be acquired in `start_run` and released in `end_run`. The `postprocess` snippet contains the bulk of the measurement code to transform the Python objects containing simulation output into a simple data structure with the results, usually a Python `dict`. This `postprocess` routine is executed once for each simulation run. Results from multiple simulation runs can be aggregated in the `TestRun` instance (described in Section 4.8).

In the simple, single-run execution flow shown in Figure 4.2, `setup` and `start_run` are run back-to-back, as are `end_run` and `postprocess`. There are several reasons why these pairs of snippets are not merged. The `setup` routine is only called once after the instance is created, whereas the `start_run` is called every time a simulation is run. This allows us to perform potentially expensive operations only once if those operations do not depend on test configuration. The reasoning for separating `end_run` and `postprocess` is similar. In the typical use case, those two snippets are always executed together. However, separation is good from an technical perspective. By separating the two, this makes it easy for the framework to execute the different steps on separate machines if needed. That is, `postprocess` and `end_run` are essentially independent and can be run concurrently. There is also another general reason for the separation of code into various snippets: to make the test writer's intent clear. The code in `postprocess` is typically more modular and reusable than the code in `end_run`, since the former contains measurements whereas the latter contains support code (e.g., clean-up code that constitutes a best practice but is not absolutely necessary). As noted above, `end_run` is not used in general; it provides an additional callback to make a particular site's environment more customizable.

4.3.2 Subtyping via Delegation

Often, we want to create derivatives of code that are mostly similar with a few differences. We may want to reuse a test in a another context (e.g., leverage an amplifier linearity test to perform a loopback test on an ADC / DAC pair[59]) or provide a custom definition of a measurement function (e.g., run an existing test but with a modified figure-of-merit). This subtyping can be implemented in several different ways, such as inheritance in class-based systems or delegation in prototype-based systems.

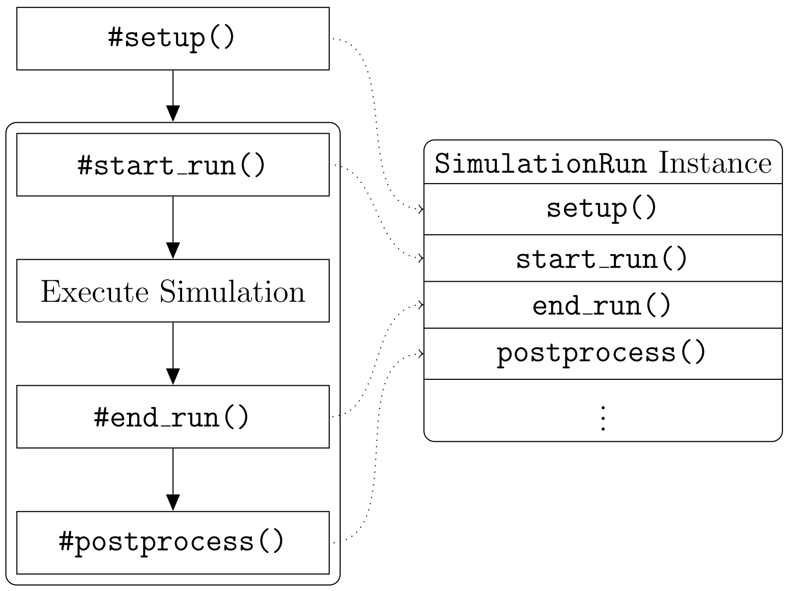


Figure 4.2: `SimulationRun` Execution Flow — The `#setup()` is executed only once when the `SimulationRun` instance is first created while the other steps are executed for every simulation.

In Python, the object oriented programming language used in CircuitBook, subtyping is usually achieved through inheritance. In earlier versions of the CircuitBook framework, inheritance was used in subtyping of `SimulationRun`. In that system, each user-provided `SimulationRun` script contained a subclass (either direct or indirect) of `SimulationRun`. A basic template for the deprecated inheritance style subtyping is shown in Listing 4.4.

A cursory look at the template does not seem very interesting; it is simply a standard Python class definition with inheritance. However, there are a couple of subtle issues with this mechanism of reuse: visibility of the parent class in scope and the use of `super`.

The template defines a class, `MySimulation`, as a subclass of the class `AnotherSimulation`. This means that the token `AnotherSimulation` has to be in the scope of the Python interpreter when the `MySimulation` class is loaded. In most Python scripts, this is not a problem; it simply means that there is an `import` statement required at the top of this script. This does pose a problem for a test

Listing 4.4: Example of Inheritance-based Subtyping

```python
class MySimulation(AnotherSimulation):
    def setup(self):
        super(MySimulation, self).setup()

        # CUSTOM CODE HERE

    def start_run(self):
        super(MySimulation, self).start_run()

        # CUSTOM CODE HERE

    def end_run(self):
        super(MySimulation, self).end_run()

        # CUSTOM CODE HERE

    def postprocess(self):
        results = super(MySimulation, self).postprocess()

        # CUSTOM CODE HERE

        return results
```

Listing 4.5: Example of Delegation-based Subtyping

```python
class MySimulation(SimulationRunProxy):
    def delegate_to:
        return 'AnotherSimulation'

    def setup(self):
        # CUSTOM CODE HERE

    def start_run(self):
        # CUSTOM CODE HERE

    def end_run(self):
        # CUSTOM CODE HERE

    def postprocess(self):
        # CUSTOM CODE HERE
```

repository. In general, we want to leverage existing tests without downloading an explicit copy and making that copy available in the local filesystem. We assume that any unresolved references to superclasses will be eventually resolved (e.g., via a lookup in the repository). Therefore, we do not want to throw any errors at load time. Instead, we want to try to locate the superclass at run-time and only throw an exception if that fails. This means that we should avoid any direct references to superclasses.

`super` is used to access the superclass of the current class. In the inheritance-based template, `super` is used to ensure that the code in the superclass is correctly executed. The `super` construct can be confusing to many novice programmers and can be a source of errors even for experienced Python programmers. In Python, `super` requires the current class to be explicitly passed. This forces the name of the class to be repeated throughout, which makes it more difficult to change. Ideally, we would like to avoid the need for `super` to make the system more accessible to users and more robust against hard-to-debug errors. Generally, errors involving inheritance mechanisms, such as `super`, are tricky, as such errors are usually very subtle. A missing `super` invocation, as used in our inheritance template, would not raise any explicit exceptions; it would simply cause some code not to be executed. This may cause the simulation results to be wrong (e.g., due to a missing stimulus) without any obvious indicators. For robustness and ease of debugging, we want the system to be fail-fast[60]. This means that we should avoid the types of subtle errors in results due to misuse of `super` as mentioned above.

By using a delegation approach to subtyping, we avoid the issues associated with inheritance-based reuse without significantly changing the mental model of the system. Instead of making `MySimulation` a subclass of `AnotherSimulation`, the `SimulationRun` that is being reused, we set `AnotherSimulation` as a delegate. In Listing 4.5, we show how to define the `MySimulation` class as delegating to the `AnotherSimulation` class in our framework. This template behaves the same as the template presented in Listing 4.4. However, it is much simpler, as it solves the issues we discussed by removing direct references to the `AnotherSimulation` class and removing `super` calls.

Consider the `#setup()` step in the `SimulationRun` execution flow as depicted in Figure 4.2. In that simplified mental model, the setup step is shown as a single piece of code that is executed. Now, let us examine this in practice using the hierarchy we described above (i.e. `MySimulation` being derived from `AnotherSimulation`). Figure 4.3 shows how the setup code at various levels of the hierarchy are executed in a classic inheritance scheme (i.e. through the use of `super`). Figure 4.4 shows the execution of setup code for the same hierarchical test construction using delegation. A linked list of the various test instances are constructed by following `delegate_to` return values. Then this linked list is used to call the `setup` routines of the various test instances starting from the top.

Delegation does not allow a class to override methods in a parent class, which breaks the model of object-oriented programming. However, this disadvantage does not apply due to the limited way we use classes. `SimulationRun` test code is declarative – when executed, it informs the CircuitBook framework what steps need to be performed to run a test but does not perform any tasks as it is executed. This means that the basic operations are idempotent (i.e., repeating the operation does not change the result). Post-processing methods are chained — the output of the parent is passed as an input to the child and ignoring that input achieves the same result as method overriding. Therefore, we do not need to override the methods in the parent class.

4.4 Stimulus Generation

In the previous section, we outlined the `SimulationRun` object that contains the main test code. Now, we will describe how our test framework supports stimulus generation, typically a major part of the `setup()` and `start_run()` snippets of `SimulationRun` scripts.

[2]When the `SimulationRun` needs to set up the test, it calls `SimulationRun#delegated_setup()` regardless of whether classic inheritance or delegation is used for subtyping. The `delegated_setup` function acts as the entry point. If there is a chain of `SimulationRunProxy` subclasses connected via their `delegate_to` functions, then it calls them each individually. Otherwise, it simply calls the appropriate `setup` function.

[3]The `pass` statement is a null operation in Python.

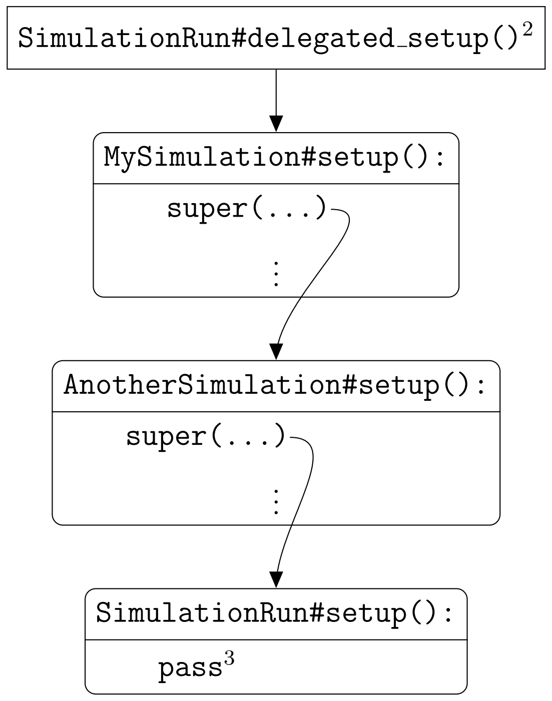


Figure 4.3: Call Flow in Inheritance-based Subtyping

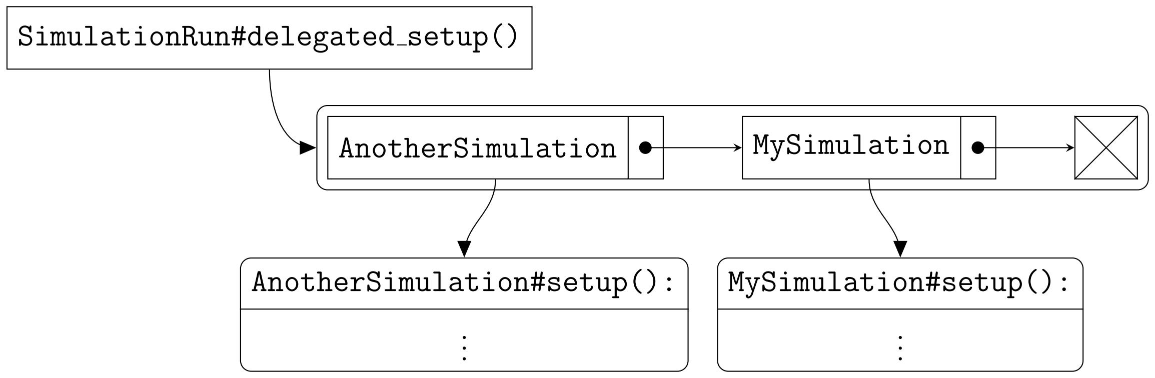


Figure 4.4: Call Flow in Delegation-based Subtyping

In order to properly evaluate circuits under test, these circuits must be appropriately driven and loaded. On a high level, this breaks into two types of tasks: test bench generation and test vector generation. First, we need to connect some circuit elements to the unit under test for it to operate properly and for the simulation to be a faithful representation of the real-world performance. These elements include power supplies, expected loading, and input drivers. Once we add these pieces, the circuit is capable of operating correctly given the correct configuration, calibration, and inputs. The second group of tasks is to generate these inputs to the system.

Note that these two types of tasks are separated based on their domains. The primary output of test bench generation is a netlist representation of the test bench. The goal of test vector generation is to create data that can be fed into the test bench. Different primitives are needed for the two groups. For the first group of tasks, we provide stimulus generators which output the appropriate SPICE commands for various sources; these are described in Section 4.4.2. The other group of tasks is to generate the vectors for testing the system and is generally done using NumPy primitives. Sometimes, as in SPICE, these two tasks are intermingled. However, the distinction is important in other contexts: in the lab, the first task relates to the connections between different test instruments and the second tasks relates to the configuration on those instruments.

Today, the user provides the test bench as a SPICE netlist, which is appended to the circuits under test and simulated. The inputs are either implicitly given by the type of source used or provided as a time-series test vector for each input required. Our goal in designing the stimulus generation system is not to replace these basic mechanisms but to provide extensions. In fact, maintaining this low-level input allows users to be productive with the framework before they learn the more advanced abstractions. This is a concept that is used in many programming environments. For example, `for` loops are sufficient for effectively manipulating data despite the availability of more powerful concepts such as iterators and collections.

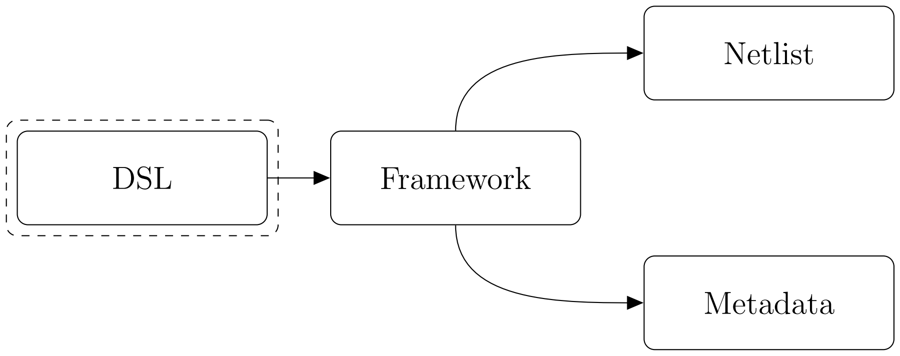
4.4.1 Test Bench Semantics

It is very useful for the test framework to understand the meaning of the circuit elements used in constructing the test bench. Users can use this data in their test scripts, and the test framework can behave more intelligently through inference. Suppose the test framework had a list of all the voltage sources used in the test bench with appropriate annotations as to the intent of these sources (i.e., power supply versus biasing). For example, we can leverage an annotated list to create a reusable power supply rejection ratio (PSRR) test. Given a base test (e.g., a working functional test that applies power, initializes with appropriate configuration, and runs some input vectors) and an output of interest, a reusable PSRR test would replace one of the power supply sources with a power supply source having noise injection capabilities and evaluate the change in output.

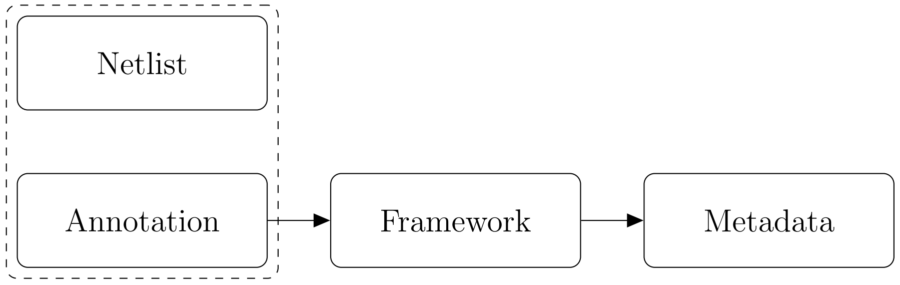
There are several ways that the test framework can create this annotated list: users can supply this information directly via annotations[61], the test framework can extract the structure by parsing the provided inputs and infer annotations automatically (e.g. using rules or machine learning techniques), or this annotated list can be created by the generator generating the test bench. These different mechanisms are shown in Figure 4.5.

No method is clearly dominant when robustness is considered. There are tradeoffs between user effort required, difficulty of learning how to use the system, and framework complexity. The relative merits of these different methods of test bench creation are summarized in Table 4.2.

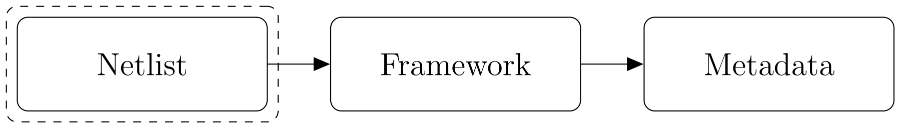
Automatic metadata inference requires the least user effort and is the easiest to learn since the user needs only to provide the netlist. The main drawback to this method is the complexity of such automatic inference, which results in robustness issues. Fundamentally, any such system requires a parser to convert the user-provided netlist into a graph and some machine learning techniques to make educated inferences about that graph. This process is statistical and occasionally will produce inaccurate metadata. When that occurs, the resulting error may be difficult to identify and troubleshoot. The problem is unavoidable because the mechanism making the error is not visible to the user. The user only has the netlist, so it is the only means for



(a) DSL-based Test Bench Creation



(b) Annotation-based Test Bench Creation



(c) Test Bench Creation with Automatic Metadata Inference

Figure 4.5: Various Ways of Creating Test Benches — The user supplies the blocks surrounded by dashed rectangles to a framework resulting in the creation of a netlist description of the test bench as well as metadata about the test bench (e.g., a list of power supplies).

Method	User Effort	Learning Required	Framework Complexity
DSL	✓✓	✓✓	✓✓
Inference	✓	✓✓	✓✓✓
Annotation	✓✓✓	✓✓✓	✓

Table 4.2: Relative Merits of Various Test Bench Creation Methods

the user to troubleshoot. The system is a blackbox, so it is hard to invert (i.e., a error at the output may have multiple causes at the input). This can quickly become an exercise in frustration for the user. We believe that predictability is important, since users need to trust their simulations and predictability contributes to that trust. Thus, for this domain, inference is not a good solution.

Annotation-based test bench creation requires the user to supply annotations with the test bench netlist. The framework required for this system can be incredibly simple, since the annotations are specified in a manner chosen by the framework's authors. For example, these annotations can be in a DSL that maps directly to the metadata, which makes the process predictable. The cost of this predictability is the effort required to annotate each test bench and learn the annotation DSL.

If we extend the DSL used in the annotation-based test bench creation to include operations that define the test bench, we can reduce the effort required to create the test bench. Instead of having the user provide a netlist and annotations and using the annotations to create metadata, we can have the user supply code in a DSL that creates both the circuit elements in the test bench and the metadata required. Obviously, this DSL will be slightly more complex than the annotation-only DSL, but by incurring this additional upfront cost, the per test bench cost is reduced. Thus, a DSL achieves the best balance of minimizing per test bench effort while maintaining predictability. Accordingly, we chose to use a DSL to create both the test bench and the needed annotations. The stimulus generators described next use this DSL.

4.4.2 Stimulus Generators

Minimally defined, a stimulus generator is a class that creates parts of a test bench when invoked. A trivial implementation of this is a class that takes no inputs in its constructor and always generates the same circuit element.

Consider a generator that produces a single 1.8 V power supply between `vdd` and `gnd`. When invoked, this generator simply injects a single voltage source into the netlist to be simulated. By using this generator instead of inserting the SPICE commands manually, we have provided the system with an annotation as to the purpose of this voltage source (i.e., it is a power supply). At simulation time, there is a list of all the stimulus generator instances available to the framework. The user and the framework can both leverage this metadata. For example, we can use this information to automatically evaluate the current draw of each power supply and produce an plot of the supply noise. We can also use this to sweep the nominal voltage of the power supply and look at the results. While such a simple generator does not save much time in terms of test bench creation, it provides a way of annotating the intent of the particular supply and it is this annotation concept that enables time-saving automation.

Practically speaking, a class as described above is not very useful since it is artificially limited in applicability (i.e., this class only works for a power supply of a particular voltage with a particular naming convention for power rails). These limitations can be addressed by parameterized generators — generators with parameterized constructors.

In fact, the CircuitBook framework provides a power supply stimulus generator that is parameterized by power supply rails and voltages called `PowerSupplyStimulus`. An example instantiation of this stimulus generator to create a single 1.8 V power supply between `vdd` and `gnd` is as follows.

```
PowerSupplyStimulus(supplies = {'vdd' : 1.8}, grounds = ['vss', 'gnd'])
```

In this example, the `vss` and `gnd` nodes are both tied together to the same negative terminal of the power supply.

While one of our main goals in using stimulus generators is to create metadata to simplify test automation, there are several additional benefits to using generators. For example, this approach is the most easily adaptable to different simulation engines. Looking at the various methods presented in Fig. 4.5, we note that stimulus generators do not require netlists unlike the other two methods. This is an important distinction because, as a result, DSL-based stimulus generators are easier to adapt to netlist-free simulation methods (e.g., Verilog, Modelica, or Simulink).

The general problem here is that we want to use a common representation for stimulus code on multiple simulation platforms. There are several ways this can be achieved, but these approaches fall into two categories: convert stimulus code to the language that the main simulator uses and then run in that simulator, or co-simulate with two different stimulation engines.

The conversion from the stimulus code to the language of the desired simulator can be done using a source-to-source compiler (i.e., a transcompiler). An example of a transcompiler is the Simulink HDL Coder[62] which converts Simulink models to Verilog or VHDL code. While, there are many previous works on manual behavior modeling of circuit elements in MATLAB / Simulink[63, 64, 65], for this application, we need automatic model generation (e.g., recent work on generating behavioral SystemVerilog models using labeled Petri nets[66]).

In theory, transcompilers solve the problem at hand. However, in practice, transcompilers can be complex and may not work well. The original stimulus code may be efficient in the source language, but, if translated, the stimulus code may not be efficient in the target language due to differences in stimulation paradigms (e.g., time-driven versus event-driven). This automatic translation may miss insights such as that a stimulus source need not be accurate at every tick of the simulator. These insights typically allow for better performance in manually ported stimulus blocks.

Instead of converting the simulation into a single language so that it can be run in a single paradigm simulation engine, co-simulation is another way to create portable stimulus code. Many of the commercially available co-simulation setups pair a higher level verification-centric language / simulator with a lower level language. For example, Simulink can co-simulate with Spectre using the Cadence Virtuoso AMS Designer

Simulator and can co-simulate with ModelSim using the Mathworks HDL Verifier.

Co-simulation avoids some of the issues faced by translation-based methods. Since it does not require code translation, a complex transcompiler is avoided. Similarly, no additional models have to be created, since the original stimulus code can be run in its own simulation engine, separate from the rest of the system. The main downside to co-simulation is the communication between simulation engines. Bi-directional information flow can be problematic. For example, it is tricky to load a circuit being simulated in SPICE with a load that is modeled in Verilog-AMS code. Successful co-simulation requires the selection of the right boundaries between blocks.

The CircuitBook simulation framework takes a translation-based approach. However, by using a higher level of abstraction, we avoid some of the complexity associated with a transcompiler. Figure 4.6 show the architecture of a theoretical netlist transcompiler (i.e., it is based on the architecture of the LLVM compiler infrastructure project[67]). Figure 4.7 shows how our framework handles stimulus generators. The obvious difference is that using a DSL avoids having a complex front-end to translate the netlist into an intermediate language since the DSL can serve that role. This means that our implementation is able to more easily produce code for different targets.

It is easy to see how the `PowerSupplyStimulus` mentioned earlier can be translated to the lab bench or a production environment. We simply produce some commands to be sent directly over a test bus (e.g., GPIB[68]) or through an common instrument interface (e.g., VISA or IVI[69]) to the lab power supplies. Similarly, we can produce equivalent output in a format for the automated test equipment in production test (e.g., OPTL[70]).

One of our goals with the CircuitBook test framework is to enable correlation of tests across various test platforms: model level, circuit level, lab test, and production test. This correlation makes it easier to trace a bug discovered in production to the root cause. Such an idea is obviously not new; it is the same reason why symbolic debuggers are more productive than machine-language debuggers. Figure 4.8 shows how the use of declarative DSL-based generators makes this correlation possible. The use of generators ensures that the same tests are being performed on the various

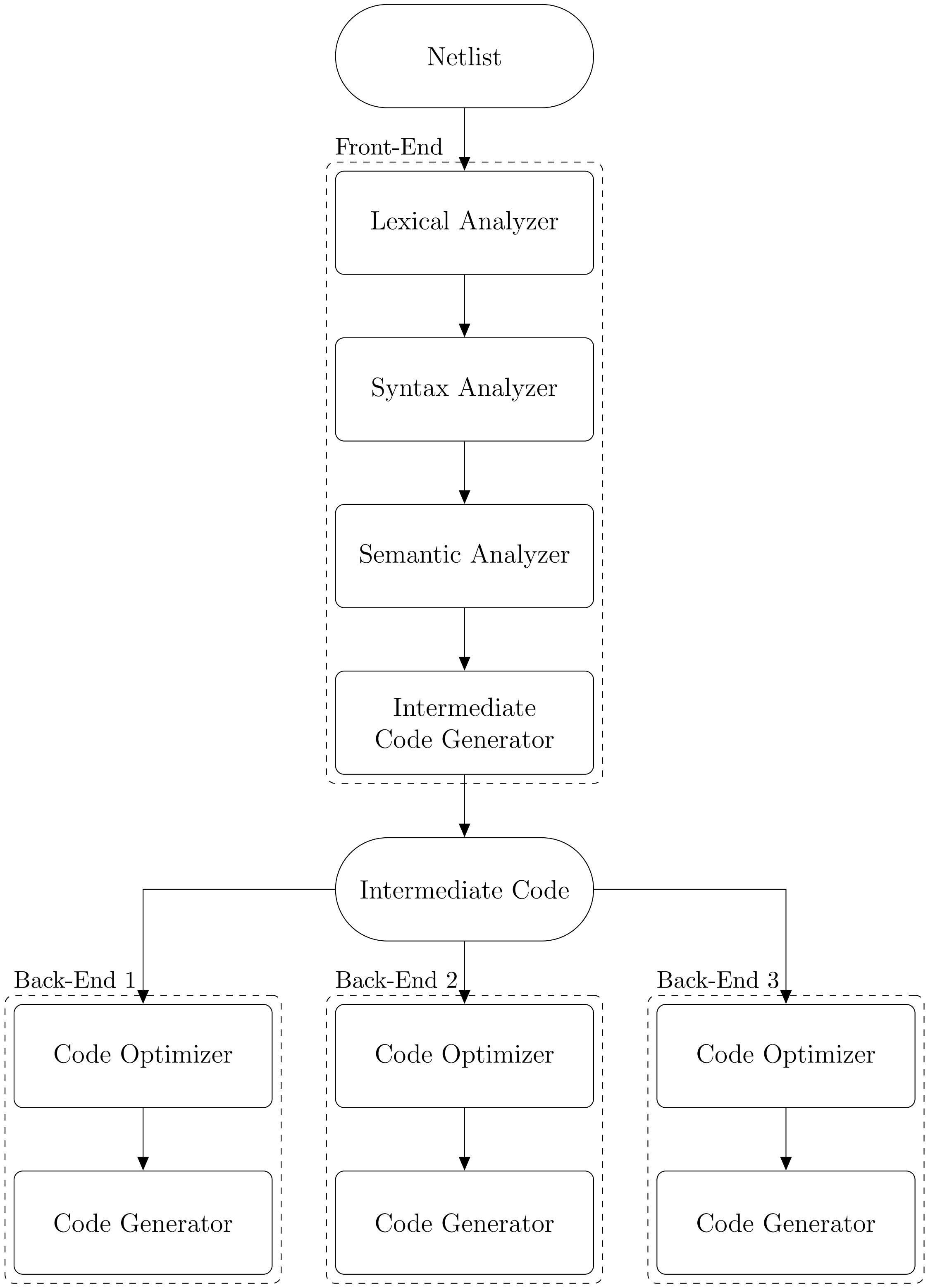


Figure 4.6: Operation of a Source-to-Source Compiler with Multiple Targets

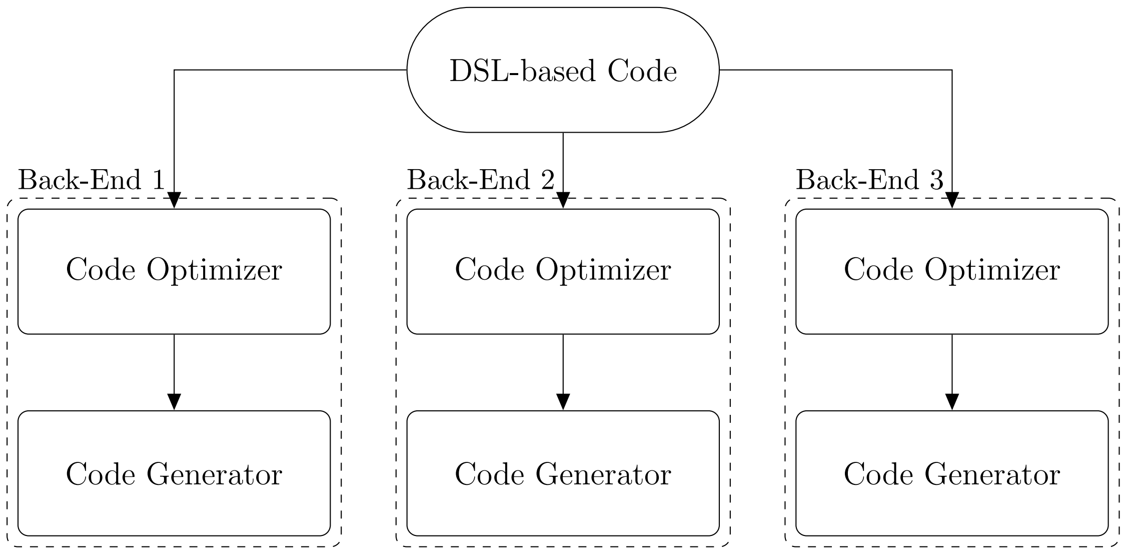


Figure 4.7: Operation of a DSL-based Code Generation System with Multiple Targets

platforms.

4.5 Simulation Execution

At the center of any test framework is the ability to actually execute the simulation. The architecture of the CircuitBook framework makes this process very straightforward. The overall execution flow for a `SimulationRun` instance is shown in Figure 4.2. In the context of that figure, we are now focusing on the *Simulation Execution* step.

First, we generate the wrappers necessary to flatten the netlist being simulated by recursively traversing through any nested interfaces and constructing the adapters, described in Section 4.1.1. Next, we elaborate all the stimulus — we call a generate function in each of the stimulus generators instantiated, passing it the current context (i.e., technology process, circuit simulator, and other environment variables). Each stimulus generator has a set of elaboration routines for each simulator it supports. If the current simulator is not supported, an error is raised. Otherwise, the underlying simulator is executed with the flattened netlist as input; this is done by generating a

[4]Standard Test Data Format

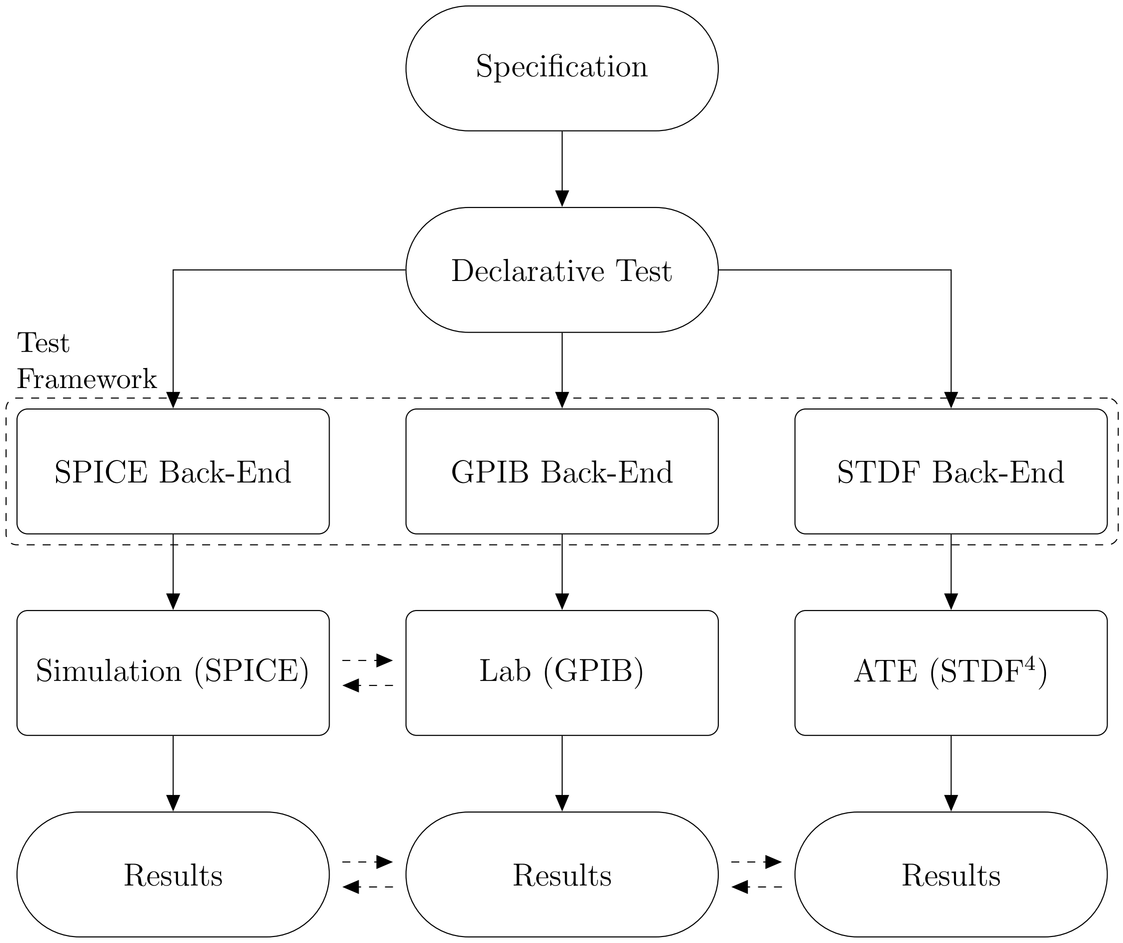


Figure 4.8: Correlation of Test Results

command line statement (e.g., `hspice test_top.sp`) and sending that to the shell.

The simulation results are read into CircuitBook data structures, which will be described in Section 4.6. Other output streams, such as logger output on `stderr`, are captured, parsed, and translated into Python exceptions in the simulation framework. This entire simulation process looks like a single function call to the user. With such an abstraction, we can transparently add features to the execution. For example, the CircuitBook framework has an optional caching feature that stores the result of simulations in a global cache, reducing the time it takes to iterate a test.

Since the test framework abstracts away the details of how the underlying simulator is called (e.g., path to the simulator binary), the actual execution process can also be modified for the needs of a particular site or group. The framework can be easily modified in the future such that simulations are parallelized to maximize throughput and minimize simulation time or placed into a queue to optimize resource usage (e.g., to minimize cost by efficiently using simulator licenses). Due to the callback style test scripts, users do not need to call any functions to invoke the simulator and so there is no visible interface that needs to be changed to enable additional functionality. For example, parallelism can be implemented by using any of the standard Python mechanisms, such as the `multiprocessing` library, in the `TestSequence` script described in Section 4.10.

4.6 Measurement

The CircuitBook test framework provides a set of analysis routines acting on the output data structures that the `SimulationRun` creates. Users can leverage these fundamental building blocks in the post processing routines of their `SimulationRun` or `TestRun` code.

As discussed in Chapter 2, we want to enable reuse by providing a common language for measurement and analysis through standardization. We first considered using MATLAB for this function, since MATLAB is the industry standard for post-processing of simulation results. Along with Simulink, MATLAB is also widely used in system-level simulations; this makes reuse of calculations and correlation of results between modeling and simulation more convenient. However, we decided against MATLAB because code organization in MATLAB is minimal and MATLAB licenses are expensive.

Instead, we provide a Python based toolkit to address some of these shortcomings, while maintaining most of the benefits MATLAB provides. In particular, we want the performance of MATLAB with better code organization and no licensing costs. To achieve this, we constructed a Python-based measurement library based on NumPy and SciPy[71]. We use NumPy as the core primarily due to its use of LAPACK[72]. Originally, MATLAB was conceived as an easy way to interact with the LINPACK[73] and EISPACK[74] linear algebra libraries. LINPACK and EISPACK have been superseded by LAPACK, which is used by modern versions of MATLAB. This means that we expected, and found in practice, that numerical performance of NumPy and MATLAB are similar.

While NumPy provides many of the basic linear algebra functions, SciPy adds additional capabilities such as regressions. The combination of NumPy and SciPy is designed to reproduce most of MATLAB's functionality with similar semantics[71]. By leveraging the NumPy and SciPy packages, we are able to focus on testing specific features rather than building fundamental math routines. In addition, the use of common math libraries prevents subtle calculation errors. That is, errors in linear algebra solvers are much harder to detect and remediate than errors in higher level functions (e.g., phase margin calculation). This is primarily due to the complexity of fast numerical routines versus the complexity of convenience functions.

4.6.1 Data Objects

Numerical data in the CircuitBook framework is stored as Python objects. There are four such fundamental data storage objects: `Vector`, `PairedVector`, `Signal`, and `Spectrum`.

The `Vector` class represents an array of numbers. Internally, the data is stored using NumPy's multidimensional arrays. By using the NumPy library to do the heavy lifting, we get memory efficiency and numerical performance similar to other computational packages (e.g., MATLAB) while keeping the class definitions for data objects simple. This abstraction hides some of the complexity of the NumPy arrays and allows the use of a different numerical back-end instead of NumPy in the future. `Vector` instances support basic arithmetic operations as well as functions that return some property about the list (e.g., max / min, argmax / argmin[5], standard deviation, or mean).

Our other data structures are internally constructed using `Vector` objects. That is, a `PairedVector` and its derivatives, `Signal` and `Spectrum`, are simply representations of a vector of ordered pairs stored as two `Vector` instances of the same length. The difference between the `PairedVector` and its derivative classes is that a `PairedVector` does not maintain any context about the units for the vectors it stores while `Signal` objects have a vector of time values (i.e., `Signal` represents a time series) and `Spectrum` objects have a vector of frequency values.

Essentially, we are introducing a notion of types to the vector of ordered pairs. This allows us to leverage type checking as a way of eliminating user errors. Note that Python is a dynamic language and does not have static types and we are using the object-oriented paradigm to provide 'type checking'. Certain methods are only defined on `Signal` instances and not on `Spectrum` instances and vice-versa. This prevents silly errors such as taking the Fourier transform of a frequency domain waveform.

All `PairedVector` objects and its derivatives have methods that provide basic manipulation such as subsetting, slicing, crossing detection, and regressions. In addition to these base methods, `Signal` objects have methods which provide integration, discrete differences, pulse measurements, frequency measurements, rise / fall time calculations, and resampling. Similarly, `Spectrum` objects have additional methods which provide gain and bandwidth calculations.

These data objects are specialized containers, rather than the general containers provided by standard programming libraries. General containers can contain any type and can be nested, whereas these typed containers can not be nested. For instance,

[5]argmax and argmin return integer indexes to the largest and smallest elements of the `Vector`, respectively.

it is possible to have an array of `Vectors` (e.g., resulting from a sweep), but a `Vector` of `Vectors` is not allowed since `Vectors` can only contain numbers.

In addition to the methods available on the instances of these data objects, these data objects enable the framework to infer some information about the intent of the user. For example, a test that involves a transient simulation and some `Spectrum` based post-processing is likely to be doing frequency domain analysis.

4.6.2 Simulation Results

After a simulation is run, the framework provides a `SimulationResult` object that holds the results of any executed analyses. This object has methods such as `#dc()` or `#tran()` that return the result corresponding to a particular analysis. Since the actual simulator output may be both large and remote (i.e., it may be on a simulation cluster), this abstraction allows us to lazily load only the results of interest as well as cache the loaded results. Each analysis is represented by a `ResultProxy` object, which is returned when the appropriate `SimulationResult` method is called. For example, a `ResultProxy` object representing the DC analysis results can be accessed in `SimulationRun#postprocess()` with `self.sim_results.dc()`.

`ResultProxy` can be accessed like a Python dictionary. Suppose we wanted to access data about a voltage source named `vbias`. This is available via the `vbias` key of the `ResultProxy` object. This mechanism can be used to access data about any visible elements – those instantiated as a part of the test bench (e.g., a stimulus generator or a load) or those declared in an interface. The data for various simulation results are provided as specialized data objects (e.g., `TransistorData`, `VoltageSourceData`, or `NodeData`). These data objects are similar in purpose and operation to `ResultProxy` (i.e., loosely wrap the underlying data and allow dictionary style access). Collectively, these result proxy objects are designed to standardize the data returned by different simulators and to provide type checking and convenience methods. Time series data are returned as `Signal` objects and frequency series data are return as `Spectrum` objects. Circuit voltages in a transient simulation can be extracted by a call to `ResultProxy#signal_for()` that returns a `Signal` with the

Listing 4.6: Example of Result Extraction in `SimulationRun#postprocess()`

```python
def postprocess(self):
    # Get the object that holds the simulation results
    acres = self.sim_results.ac()

    # Create the output transfer function as a spectrum
    output = acres.spectrum_for('outdiff').magnitude()

    # Extract the DC gain and -3 dB bandwidth from the transfer function
    gain = output.dc_gain()
    bw = output.bandwidth(1 / sqrt(2))
```

appropriate node voltages and time steps. There is a `ResultProxy#spectrum_for()` that performs a similar function for AC simulations. These functions make it easy for the user to retrieve data in the form of data objects.

Listing 4.6 shows an example of a `SimulationRun#postprocess()` that extracts the gain and bandwidth of an output. First, we get a `ResultProxy` object representing the AC analysis results and store that in `acres`. Then we call the `spectrum_for` method on that result object to get a `Spectrum` object of the waveform of the node `outdiff`. We transform it with a call to `magnitude()`, which converts a complex `Spectrum` into a real `Spectrum` of the magnitude, before storing it into `output`. Finally, we extract the DC gain and -3 dB bandwidth by calling the appropriate methods defined on the `Spectrum` object.

4.7 Logging with `Epilog`

Ultimately, simulation results are used to make decisions. Some of these decisions are made by users and some are made algorithmically. The use of structured data (i.e., data objects and result proxy objects) make it easy for machine consumption of simulation results. Clearly, there is a similar need for human readable data. In the CircuitBook framework, we address this need through the `Epilog` module, a set of library functions and associated tools that focus on producing data for human analysis.

There are many stakeholders in any integrated circuit design project and these various stakeholders are interested in different projections of the underlying simulation results. Furthermore, the reporting needs of a particular stakeholder changes over time based on the current stage of the design. To serve these diverse needs, a test framework should provide reliable summaries that faithfully represent the underlying data while providing access to the data when necessary.

In the initial phase of crafting a circuit, designers need access to the detailed information about the particular block being designed. This may mean that the simulation report should contain the cross product of test vectors with measurements (i.e., everything). At this stage, the designer needs and wants to look at all the measurements to identify any unexpected behavior, which by definition cannot be automatically identified. As a design gets more stable, the designer may want to reduce the information that needs to be reviewed by summarizing. For example, in the initial stage of a link design, the actual eye diagram needs to be examined but later on, only the specifications of that eye diagram (e.g., vertical eye opening, jitter measurements, quality factor) need to be checked. Once a design is mostly finalized and being tweaked, the designer may be instead interested in the changes in various specifications across iterations. From a project management or testing perspective, the data of interest is whether specifications are met by a block, that is, a checklist of pass or fail marks.

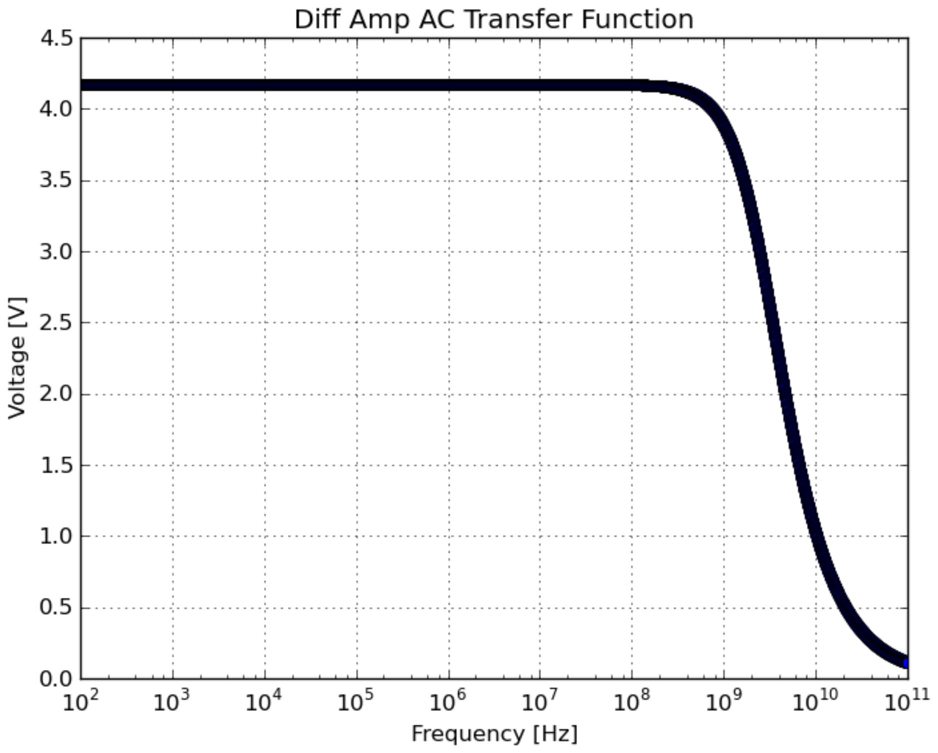
These needs are very similar to those faced by users of software logs[75]. The different detail levels of simulation results required are analogous to the verbosity levels in software logs. Software logs are often neglected because they are sometimes perceived to be secondary to the main function of the software, and this neglect often makes troubleshooting a nightmarish task. Similarly, the outputs of simulations are often neglected. The typical work-product of a set of tests may be a few values and some messages printed to the consoles and the raw simulation outputs. This may work fine when the test writer initially uses the test because the entire context is available. The lack of documentation slowly becomes a problem over time. It is often unclear whether the printed values are the most important metrics or simply what the test writer was most recently tuning. The messages printed may be for debugging purposes or actual errors (i.e., violations in some assumptions). Even messages specifically labeled as errors may not be so serious. Often, certain error messages are expected and ignored during some non-normal operating conditions such as start up. The problem can be even worse when dealing with the raw simulation output. There may be little to no documentation as to which nodes to probe and when those nodes need to be observed.

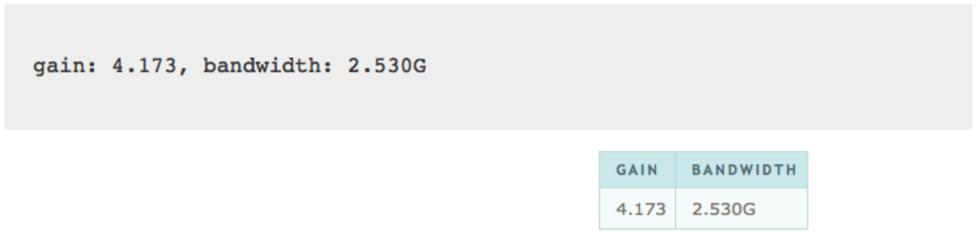
As elsewhere in the CircuitBook framework, we apply the same techniques and principles to address this issue of ad-hoc reporting without documentation of intent. We invert the problem. Instead of trying to figure out the nodes of interest in a large waveform dump, we force the users to identify the nodes of interest and only expose those nodes. Since the measurements are written using our library of functions and the data is stored in our data objects, we have a lot of metadata about the intent of the user and the nature of the data that can be leveraged to automate and simplify reporting. For example, there are plotting methods attached to `Signal` and `Spectrum` with reasonable defaults that produce nice looking graphics out of the box.

`Epilog` is essentially a DSL for writing test reports. Listing 4.7 shows a small example of an `Epilog` snippet that plots a `Spectrum` object and emits some summary information. It highlights some of the basic functionality (i.e., plots and tables) provided by the libraries. When the test is executed some of this information is output to the standard output and a report is generated and saved to disk. The verbosity of the report can be customized in the test script by changing `Epilog` parameters. In our current implementation the report is produced in HTML, but due to the modular design, other output formats should also be easy to implement. An example of the output corresponding to Listing 4.7 is shown in Figure 4.9.

4.8 `TestRun`

In the previous section, we focused on how `SimulationRun` objects capture the test collateral for a single simulation. Now, we will look at `TestRun` objects that control the execution of `SimulationRun` scripts and aggregate measurement results and `Epilog` reports across those executions. Listings 3.4 and 4.8 show examples of `TestRun` scripts.





```
gain: 4.173, bandwidth: 2.530G
```

GAIN	BANDWIDTH
4.173	2.530G

Figure 4.9: Screenshot Showing an Example of `Epilog` Output

Listing 4.7: Example of `Epilog` that Plots a `Spectrum` and Produces a Summary Table

```
# Create a new Epilog section
with self.epl_section() as epl:
    # Plot the transfer function
    output.plot(ylabel='Voltage [V]',
                title='Diff Amp AC Transfer Function',
                filename=epl.imagefile('outdiff_vs_freq'))

    # Emit a summary line
    epl.println("gain: %.3f, bandwidth: %.3fG" % (gain, bw/1e9))

    # Add a summary table
    with epl.table() as t:
        with t.header_row() as r:
            r.add_cell('Gain')
            r.add_cell('Bandwidth')
        with t.row() as r:
            r.add_cell('%.3f' % gain)
            r.add_cell('%.3fG' % (bw / 1e9))
```

We use `TestRun` objects to encapsulate the less reusable parts of the test collateral. As we discussed previously in Chapter 3, some parts of a test setup are specific to a particular circuit instance (e.g., an industrial part has different temperature ranges for simulation compared to a MIL-SPEC part). If we only had `SimulationRun` objects, then the user needs to make a choice between writing reusable tests or writing practical tests. Clearly, test reuse does not have a chance in such a contest.

`TestRun` scripts are primarily used to run multiple `SimulationRun` or run a single `SimulationRun` multiple times. These scripts rely on native Python constructs for control flow. Examples of uses include running a simulation over process corners, automatic circuit optimization, and running interconnected simulations (e.g., a PLL simulation generally requires some block level simulation to extract parameters such as K_{vco} and then running system level simulations with those parameters). `TestRun` are connected to a single interface which is bound to a circuit at runtime.

The conceptual model for `TestRun` is analogous to that of `SimulationRun` as described in Section 4.3 and its execution flow, shown in Figure 4.10, is similar to the `SimulationRun` execution flow (Figure 4.2). In each `TestRun` script, the user

Listing 4.8: Example of `TestRun` — We use `self.construct('PllPNoiseSimulation')` to build an instance instead of the normal `PllPNoiseSimulation()` constructor call to allow for deferred loading of `PllPNoiseSimulation`. This allows the system to load `PllPNoiseSimulation` before `PllPNoiseTest` is in scope; this is described in Section 4.3.2.

```
class PllPNoiseTest(TestRun):
    def setup(self):
        model = PTM065ProcessModel(scale=0.035e-6)
        self.pn = self.construct('PllPNoiseSimulation', model)
        self.ref_freq = 1e9
        self.div_ratio = 2

    def test(self):
        """ Perform phase noise simulation"""
        for corner in self.pn.corners():
            print 'Performing %s corner simulation' % corner

            # Set some variables in the simulation
            self.pn.set_variable('vdd_noisevector','[1k 1e-20 1M 1e-20 
                100M 1e-20]')
            self.pn.set_variable('ref_freq', self.ref_freq)
            self.pn.set_variable('div_ratio', self.div_ratio)

            # Execute the simulation
            self.simulate(self.pn, corner)
```

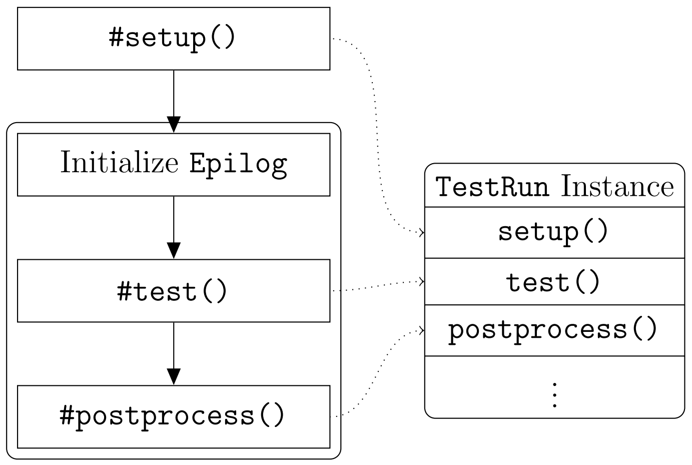


Figure 4.10: `TestRun` Execution Flow

provides code for the `setup()`, `test()`, and `postprocess()`. The `setup()` snippet is usually used to configure the technology process. `test()` is the main snippet that contains calls to execute different `SimulationRun` scripts. `postprocess()` is optional and allows results from multiple `TestRun` scripts to be collected in a `TestSequence`, discussed in Section 4.10.

Users call `TestRun#construct` with the name of a `SimulationRun` subclass as a string to get an instance of that class. For example, to get an instance of the `MySimulation` class, a user calls `TestRun#construct('MySimulation')`.[6]

4.9 Test Loading and Execution

In the previous sections, we discussed the `SimulationRun` and `TestRun` scripts written by users. In this section, we look at how the framework runs those tests.

In our system, we provide a command-line executable that automatically runs a `TestRun` subclass by calling the user-provided callback at the appropriate time. The user runs a test by executing the following command at a shell.

[6]This is done in lieu of the normal constructor method (i.e., `MySimulation()`) because the `MySimulation` class may not be in scope when the calling `TestRun` is loaded since the dynamic loading order is arbitrary. Being agnostic to the load order avoids a dependency resolution problem, as there may be circular dependencies.

```
run.py MyTest.py my_circuit
```

The example shown above will cause the test framework to read the file `MyTest.py` and run all the `TestRun` subclasses contained within against a circuit called `my_circuit`. Now, let us take a look at this is achieved.

4.9.1 Dynamic Test Class Loading

While our example shows a file being loaded, the `run.py` command accepts any resource (i.e., a file, an URL to a file, or a name of a class in the repository). We grab the code from the resource and dynamically load that code into the running Python instance. The dynamic loading is done by leveraging the Python import internals as exposed via the `imp` module in Python — we call `imp.load_source` to load the source file and it returns a handle to the module. This is handled by our `ObjectLoader` class.

We use Python's dynamic introspection abilities to examine the loaded module for classes of interest. In Python, `dir()` returns a list of names defined by a module, `__getattribute__()` returns a reference to an object by its name as a string, and `inspect.isclass()` tests an object to see if it is a class. By combining these features, we are able to get a list of all the classes in the module that we dynamically loaded (i.e., use `dir()` to get a list names in the module, use `__getattribute__()` to convert those names into object references, and then check if each name refers to a class via `inspect.isclass()`).

Now that we have a list of classes, we iterate through them to find subclasses of `TestRun` and `SimulationRun`. We store a pointer to each of the `SimulationRun` classes indexed by its name for later use in a class cache. For each of the `TestRun` objects, we call its `run` function with the name of the circuit to test. This `run` function is usually not extended by the user and resolves to `TestRun#run` (i.e., the same function on the framework provided base class). In `TestRun#run`, we essentially call `setup()`, `test()`, and `postprocess()` on the user-provided `TestRun` subclass in order.

4.9.2 Circuit / Interface Loading

In order to provide the `SimulationRun` class with the appropriate circuits and interfaces, we need to load those resources either from the local file system or the repository. Since these are simply text files, the actual loading process is simple. The main challenge here is dependency resolution. Circuits can connect to multiple interfaces and each interface can also adapt to other interfaces higher in the hierarchy. Circuits can also instantiate other subcircuits through interfaces. When loading a circuit, we also need to load all the circuits and interfaces that it relies upon. This is an instance of a dependency resolution problem.

The dependency requirements can be viewed as a directed acyclic graph (DAG), and dependency resolution is simply a topological sort of this graph — this can be visualized as all the edges of the DAG pointing in the same general direction. Once this sort is complete, the graph provides the order in which resources need to be loaded. The general problem can be complicated to perform efficiently for large graphs or version ranges (i.e., the version of a dependency requirement accepts a range of values), but it is actually relatively simple for us, since the number of resources is small, and we do not allow version ranges in interface specifications. We implement this by chasing down all the dependencies via a depth first search while keeping track of visited vertices. We use `NetworkX`[76], a Python library for working with complex networks, to store the underlying graph data structures.

4.10 `TestSequence`

The combination of `TestRun`, `SimulationRun`, and `run.py` allows the user to write and execute tests against a single circuit. Users may want to run tests against different circuits and aggregate the results in a useful way. A common example of this is comparing performance between iterations of a circuit. Users wanted to leverage the graphing and data presentation facilities provided by Epilog in comparing different iterations of a circuit.

While this can be achieved using only the constructs we have mentioned above, it

Listing 4.9: Example of `TestSequence`

```python
class MultiDiffAmpTestSeq(TestSequence):
    def sequence(self):
        d1 = self.execute('DiffAmpTest', 'diffamp1')
        d2 = self.execute('DiffAmpTest', 'diffamp2')

        # Compare results here...
        print d1['power'], d2['power']
```

is not very user friendly. Users had to use shell scripts to execute a `TestRun` multiple times against different circuits and then export and process that data. This seemed reminiscent of the ad-hoc testing method that we discussed in Chapter 2. This motivated us to add a `TestSequence` class to provide additional flexibility. `TestSequence` avoids this by allowing the user to execute `TestRun` scripts from within Python. It is essentially a programmatic interface (i.e., an API) to the internal framework components that load and execute tests. An example is shown in Listing 4.9.

Both the concept and implementation of `TestSequence` are very simple. There is a single `sequence()` method where users can provide code, and that method is executed when the `TestSequence` is provided as an argument to `run.py`, the command-line tool discussed in Section 4.9.

4.11 Repository

The first step to reusing something is finding it, so a system for locating circuits and tests is a natural companion to a test framework. We constructed the CircuitBook repository to serve as a companion to the CircuitBook test framework. This repository stores and indexes circuit instances, interfaces, and test classes. It allows the user to quickly find a test to use in or adapt to a particular situation.

In Section 3.8, we discussed the hierarchical classification system for organizing circuits and tests. It is implemented using the Ruby-on-Rails framework as a web application. Data is stored in a combination of a git repository and a MySQL database.

The test collateral is stored in the git repository for traceability (i.e., nothing is overwritten); it also allows a quick way for the circuits and tests to be backed up. The database handles the metadata for searching and indexing.

In addition to search features, the repository also organizes circuits and results generated by the `Epilog` module (described in Section 4.7).

4.11.1 Metadata Extraction

One of the key features of the repository is to present the user with a window into the relationships between test collateral. In order to do this, we must extract metadata from the test files. As shown in Figure 3.6, we provide diagrams of relationships between test classes and with circuit interfaces. When a test class is loaded into the repository, we load it through dynamic loading mechanisms similar to that described in Section 4.9. Once this is done, we use the built-in `ast` module in Python to access the abstract syntax tree (AST) of the test code. The AST is a graph representation of all the functions, expressions, and statements in the code, where each element is represented as an object. With this AST, we are able to derive meaning from the code.

To extract relationships, we parse through the AST and look for calls to a function called `construct`. For each call, we verify that the object it is being called on is a derivative of `TestRun` in case the user has a `construct` function. Then, we examine the arguments passed to these `TestRun#construct` calls, and this tells us which `SimulationRun` was instantiated in a particular test. This information is output as metadata and stored in our database. When a user browses a test in the repository, we render the DAG of the relationships using Graphviz[77]. Laying out graphs so that they are visually pleasing is a long-studied problem with a complex solution. We leverage the many decades of research into this by using Graphviz, a graph drawing program.

In additions to relationships, we can extract other information about a test, such as the analysis types used and the stimulus generators invoked, through the AST. This metadata is used to enable help users find a test to reuse and is described in

Section 3.8.3.

4.12 Summary

In this chapter we explored the test framework in detail to show how each component work and how these components come together. First, we showed how to specify circuits and interfaces. Then, we examined the various classes (e.g., `SimulationRun`, `TestRun`) that hold the test scripts. Next, we looked at how the `run.py` script runs and executes the test scripts. Finally, we discussed how the repository extracts metadata from tests.

Chapter 5

Conclusion

Tests play an important role in the circuit design process, which means that the collateral files generated to perform the test have significant value. These files codify some of the requirements for the circuit being tested. Given its importance, test collateral should be archived in a way that facilitates reuse. Unfortunately, given the ad-hoc nature of test generation, there is neither a discipline to archive these tests nor a set of standard interfaces/sockets to plug saved tests into.

To address this issue, we separate each test into its constituent parts: circuit representation, simulation directives, stimuli, measurement, and results analysis routines. These components have different natural expression forms. For example, stimuli are described by a time-series, whereas post-processing is a function that takes a time-series and returns some parameters of interest. We provide a different DSL to allow the user to better express the description of each block. Once we separate the test code in this way, natural interfaces appear between these components (e.g., the interface of a stimulus is that it exports a time-series). This decomposition was presented in Section 3.1 and shown in Figure 3.1. We believe that this decomposition separates the reusable and less reusable components of a test to make it easy to reuse.

Our use of interfaces to decouple tests and circuits resulted in a cleaner separation of test code and circuits. It is easy to determine a test's goals from looking only at the test code and the associated interface without actually looking at the specifics of the circuit. This suggests that circuit interfaces effectively abstract the details of the underlying circuit. In ad-hoc test generation, users often access internal node voltages and currents and device operating points without any explicit declaration. This kind of cheating can make it difficult to understand the operation of a test. For example, a test for an differential amplifier may want to check the tail current. In an ad-hoc test this may simply be specified as `I(M15)` where `M15` is the tail device. It is difficult to surmise what `M15` refers to from only looking at the test code. In a CircuitBook test, this device needs to be declared at the interface and hopefully it is declared with a meaningful name (e.g., Mtail). This adds a little overhead to the testing process — choosing the appropriate abstraction (i.e., what information and wires to expose and how to name them) for a circuit takes effort.

We developed the CircuitBook test framework to allow users to write test code using our partitioning and run tests constructed from those building blocks. In our experience with the tool and from talking to users, our test framework appears to deliver on its promises — there is minimal code duplication and it is easy to make derivative tests. After building a few tests, users seem to have little trouble with understanding the value of the test framework and constructing complex reusable tests with limited guidance.

Users did not use multiple-level hierarchical interfaces – most users used our test framework for a single project and did not have a need for multiple levels of hierarchy. However, two-level hierarchical interfaces were used to perform comparisons between different versions of a circuit.

As mentioned in Chapter 4, we used `SimulationRun`, `TestRun`, and `TestSequence` classes to separate the different code blocks in a test. This separation worked well and was picked up by users surprisingly easily. We thought there would be some confusion on where various parts of the test should be written. However, in practice, this was not that difficult. Users placed the majority of the test code in the `SimulationRun` scripts and only used `TestRun` and `TestSequence` as necessary to aggregate results across `SimulationRun` scripts and bind technology processes. This is the desired behavior, as the `SimulationRun` classes are the most reusable.

Overall, the DSLs we provided were not too difficult to learn. The biggest barrier in terms of learning to write test code (i.e., not including the choice of an appropriate interface) was learning Python. We started this project with a belief that Python was a very popular programming language, especially amongst the scientific and engineering communities. While we still believe that to be true, we were surprised at the lack of adoption in our limited sampling of the circuit design community. Many users did not want to expend the effort to learn a new language regardless of the actual difficulty; the perceived difficulty may have been a bigger barrier than the actual difficulty.

5.1 Future Work

We have found that one of the main challenges with the CircuitBook test framework has been convincing users to adopt the system. We believe this can be attributed to the initial learning required to be productive. The CircuitBook test framework does not significantly speed up the time required to make a new test for first time when the time to learn the framework is included. The productivity gains come from reusing the resulting test collateral for future variants. Users are often concerned more about the task at hand than future benefits, so our current test framework may not be attractive to time-constrained designers.

The user onboarding challenge can potentially be addressed in several ways: reduce the effort to required to be minimally productive and facilitate learning and debugging.

1. For a prospective user, learning our test framework and how to use it to create a working test from scratch will take more effort than building the same test in the current ad-hoc methodology. If we fill our repository with reusable test components, then the effort to build the first test from reusable components may be less than building the test in an ad-hoc manner. Obviously, it is challenging to build a comprehensive collection of reusable test components.

   Instead of such a gargantuan task, it may be sufficient to focus on a particular class of circuits (i.e., a branch on the hierarchy) and provide building blocks that allow users to quickly construct tests for that circuit class. For example, we can provide building blocks for testing amplifiers or data converters.

2. We provide a reference manual that shows the arguments of each library function, what that function returns, and a basic description. It may be useful to expand this such that there is a link to an example code snippet showing that function in use. Doing so may help to provide users context and allow them to better understand how these functions can be composed to build larger blocks.

3. Our error handling can be improved to be more robust. Because Python is a dynamic language, a simple mistake can be parsed as a "valid" program that creates an error at runtime. This means that users might become frustrated by a long stack traceback. It may be worthwhile to build a lint type tool that can quickly check a test script to see if it violates any best practices or looks suspicious.

While we believe the CircuitBook test framework is ready for prime-time, these suggestions will make it more user friendly.

Bibliography

[1] Ravi Subramanian. Verification of nanometer mixed-signal ICs with the Analog FastSPICE platform. Berkeley Wireless Research Center Seminar, 2009.

[2] MATLAB. *Version 7.10.0 (R2010a).* The MathWorks Inc., Natick, Massachusetts, 2010.

[3] Simulink. *Version 7.5.* The MathWorks Inc., Natick, Massachusetts, 2010.

[4] Mathcad. *Version 15.0.* Parametric Technology Corporation, Needham, Massachusetts, 2010.

[5] Virtuoso Analog Design Environment. *Version 6.1.* Cadence Design Systems, Inc., San Jose, California, 2009.

[6] T. Quarles. The SPICE3 implementation guide. Technical Report UCB/ERL M89/44, EECS Department, University of California, Berkeley, 1989.

[7] Spectre Circuit Simulator. *Version 5.0.* Cadence Design Systems, Inc., San Jose, California, 2003.

[8] Galaxy Custom Designer. *Version 2009.06.* Synopsys, Inc., Mountain View, California, 2009.

[9] *AMSUltra: Virtuoso AMS Simulator with Virtuoso UltraSim FastSPICE Solver.* Cadence Design Systems, Inc., 2004.

[10] Patrick Kuanlye Goh. *A fast multi-purpose circuit simulator using the latency insertion method.* PhD thesis, University of Illinois at Urbana-Champaign, 2012.

[11] K.S. Kundert. Introduction to RF simulation and its application. *Solid-State Circuits, IEEE Journal of*, 34(9):1298 –1319, sep 1999.

[12] Bob Gautier, Chris Loftus, Edel Sherratt, and Lynda Thomas. Tool integration: experiences and directions. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 315–324, New York, NY, USA, 1995. ACM.

[13] F. Silveira, D. Flandre, and P.G.A. Jespers. A gm/ID based methodology for the design of CMOS analog circuits and its application to the synthesis of a silicon-on-insulator micropower OTA. *Solid-State Circuits, IEEE Journal of*, 31(9):1314 –1319, sep 1996.

[14] D. Flandre, A. Viviani, J.-P. Eggermont, B. Gentinne, and P.G.A. Jespers. Improved synthesis of gain-boosted regulated-cascode CMOS stages using symbolic analysis and gm/ID methodology. *Solid-State Circuits, IEEE Journal of*, 32(7):1006 –1012, jul 1997.

[15] G.G.E. Gielen, H.C.C. Walscharts, and W.M.C. Sansen. ISAAC: a symbolic simulator for analog integrated circuits. *Solid-State Circuits, IEEE Journal of*, 24(6):1587–1597, dec 1989.

[16] A. Doboli and R. Vemuri. Behavioral modeling for high-level synthesis of analog and mixed-signal systems from VHDL-AMS. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 22(11):1504 – 1520, nov. 2003.

[17] R.A. Rutenbar, G.G.E. Gielen, and J. Roychowdhury. Hierarchical modeling, optimization, and synthesis for system-level analog and RF designs. *Proceedings of the IEEE*, 95(3):640 –669, march 2007.

[18] Waisum Wong, Xiaofang Gao, Yang Wang, and S. Vishwanathan. Overview of mixed signal methodology for digital full-chip design/verification. In *Solid-State and Integrated Circuits Technology, 2004. Proceedings. 7th International Conference on*, volume 2, pages 1421 – 1424 vol.2, oct. 2004.

[19] M. Horowitz, M. Jeeradit, F. Lau, S. Liao, B. Lim, and J. Mao. Fortifying analog models with equivalence checking and coverage analysis. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 425 –430, june 2010.

[20] S. De Sirkar, R. Tupuri, C.S. Madhuri, G. Rajagopalan, K.D. Mehta, C.G. Madhukar, and G. Bajpe. Design migration across technology - making it work. In *VLSI Design, 1992. Proceedings., The Fifth International Conference on*, pages 68 –72, jan 1992.

[21] M. Knieser, M. Lucak, F. Wolff, and C. Papachristou. SoC gate level design migration. In *ASIC/SOC Conference, 2002. 15th Annual IEEE International*, pages 155 – 159, sept. 2002.

[22] K. Francken and G. Gielen. Methodology for analog technology porting including performance tuning. In *Circuits and Systems, 1999. ISCAS '99. Proceedings of the 1999 IEEE International Symposium on*, volume 1, pages 415 –418 vol.1, jul 1999.

[23] C.A. Makris and C. Toumazou. Analog IC design automation. II. Automated circuit correction by qualitative reasoning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 14(2):239 –254, feb 1995.

[24] Lou Prado Kazuhiro ODA and Anthony J. Gadient. A new methodology for analog/mixed-signal (AMS) SoC design that enables AMS design reuse and achieves full-custom performance. In *The Ninth IEEE/DATC Electronic Design Processes Workshop. EDP-2002*, 2002.

[25] S. Hammouda, H. Said, M. Dessouky, M. Tawfik, Q. Nguyen, W. Badawy, H. Abbas, and H. Shahein. Chameleon ART: a non-optimization based analog design migration framework. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 885 –888, 0-0 2006.

[26] T. Massier, H. Graeb, and U. Schlichtmann. The sizing rules method for CMOS and bipolar analog integrated circuit synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(12):2209 –2222, dec. 2008.

[27] Yi-Peng Weng, Hung-Ming Chen, Tung-Chieh Chen, Po-Cheng Pan, Chien-Hung Chen, and Wei-Zen Chen. Fast analog layout prototyping for nanometer design migration. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 517 –522, nov. 2011.

[28] M.G.R. Degrauwe, O. Nys, E. Dijkstra, J. Rijmenants, S. Bitz, B.L.A.G. Goffart, E.A. Vittoz, S. Cserveny, C. Meixenberger, G. van der Stappen, and H.J. Oguey. IDAC: an interactive design tool for analog CMOS circuits. *Solid-State Circuits, IEEE Journal of*, 22(6):1106 – 1116, dec 1987.

[29] H.Y. Koh, C.H. Sequin, and P.R. Gray. OPASYN: a compiler for CMOS operational amplifiers. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 9(2):113 –125, feb 1990.

[30] F. El-Turky and E.E. Perry. BLADES: an artificial intelligence approach to analog circuit design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(6):680 –692, jun 1989.

[31] R.R. Neff, P.R. Gray, and A. Sangiovanni-Vincentelli. A module generator for high-speed CMOS current output digital/analog converters. *Solid-State Circuits, IEEE Journal of*, 31(3):448 –451, mar 1996.

[32] G. Jusuf, P.R. Gray, and A.L. Sangiovanni-Vincentelli. CADICS-cyclic analog-to-digital converter synthesis. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 286 –289, nov 1990.

[33] R. Harjani, R.A. Rutenbar, and L.R. Carley. OASYS: a framework for analog circuit synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(12):1247 –1266, dec 1989.

[34] E.S. Ochotta, R.A. Rutenbar, and L.R. Carley. Synthesis of high-performance analog circuits in ASTRX/OBLX. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(3):273 –294, mar 1996.

[35] R. Phelps, M. Krasnicki, R.A. Rutenbar, L.R. Carley, and J.R. Hellums. Anaconda: simulation-based synthesis of analog circuits via stochastic pattern search. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(6):703 –717, jun 2000.

[36] G.G.E. Gielen and R.A. Rutenbar. Computer-aided design of analog and mixed-signal integrated circuits. *Proceedings of the IEEE*, 88(12):1825 –1854, dec 2000.

[37] G.G.E. Gielen and J.E. Franca. CAD tools for data converter design: an overview. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 43(2):77 –89, feb 1996.

[38] G. Van der Plas, G. Debyser, F. Leyn, K. Lampaert, J. Vandenbussche, G.G.E. Gielen, W. Sansen, P. Veselinovic, and D. Leenarts. AMGIE - A synthesis environment for CMOS analog integrated circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(9):1037 –1058, sep 2001.

[39] John Crossley, Hanh-Phuc Le, Rachel Nancollas, Alberto Puggelli, and Elad Alon. Berkeley Analog Generator. Poster, May 2012.

[40] The MathWorks Inc., Natick, Massachusetts. *MATLAB Programming Fundamentals - R2012a*, 2012.

[41] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 47–52, New York, NY, USA, 2010. ACM.

[42] G. Lawton. LAMP lights enterprise development efforts. *Computer*, 38(9):18 – 20, sept. 2005.

[43] The MathWorks Inc., Natick, Massachusetts. *Aerospace Blockset User's Guide*, Version 1 edition, 2004.

[44] T.J. Barnes. SKILL: a CAD system extension language. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 266 –271, jun 1990.

[45] OCEAN Reference. *Product Version 5.1.41.* Cadence Design Systems, Inc., San Jose, California, 2004.

[46] Dinu Gherman. Empy, a python templating system. Presented at EPC 2003, European Python and Zope Conference in Charleroi, Belgium, June 2003.

[47] Synopsys, Inc. *HSPICE Reference Manual: Commands and Control Options*, Version B-2008.09 edition, 2008.

[48] Mentor Graphics Corporation. *Eldo User's Manual*, Software Version 6.6_1 Release 2005.3 edition, 2005.

[49] J.A. Mielke. Frequency domain testing of ADCs. *Design Test of Computers, IEEE*, 13(1):64 –69, spring 1996.

[50] M.F. Wagdy and S.S. Awad. Determining ADC effective number of bits via histogram testing. *Instrumentation and Measurement, IEEE Transactions on*, 40(4):770 –772, aug 1991.

[51] M.F. Wagdy and M. Goff. Linearizing average transfer characteristics of ideal ADC's via analog and digital dither. *Instrumentation and Measurement, IEEE Transactions on*, 43(2):146 –150, apr 1994.

[52] IEEE standard for terminology and test methods for analog-to-digital converters. *IEEE Std 1241-2010 (Revision of IEEE Std 1241-2000)*, pages 1–139, 14 2011.

[53] T.E. Linnenbrink, S.J. Tilden, and M.T. Miller. ADC testing with IEEE Std 1241-2000. In *Instrumentation and Measurement Technology Conference, 2001. IMTC 2001. Proceedings of the 18th IEEE*, volume 3, pages 1986–1991 vol.3, 2001.

[54] W. Kester. *AN-215 — Designer's Guide to Flash-ADC Testing*. Analog Devices, Inc., 1990.

[55] Francesco Ricci and Bracha Shapira. *Recommender systems handbook*. Springer, 2011.

[56] Caitlin Sadowski and Greg Levin. Simhash: Hash-based similarity detection. Technical report, Technical report, Google, 2007.

[57] S. Sikand. High performance scalable hardware configuration management. 2003.

[58] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. YAML Aint Markup Language (YAML) Version 1.2, 2009.

[59] Xuan-Lun Huang and Jiun-Lang Huang. ADC/DAC loopback linearity testing by DAC output offsetting and scaling. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(10):1765–1774, 2011.

[60] J. Gray. Why do computers stop and what can be done about it? Technical Report TR 85.7, Tandem Computers, 1985.

[61] Dean Liu. *A Framework for Designing Reusable Analog Circuits*. PhD thesis, Stanford University, 2003.

[62] *Integrating Xilinx System Generator with Simulink HDL Coder*. The MathWorks Inc., 2008.

[63] K.W. Current, J.F. Parker, and W.J. Hardaker. On behavioral modeling of analog and mixed-signal circuits. In *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on*, volume 1, pages 264–268 vol.1, oct-2 nov 1994.

[64] V.F. Pires and J.F.A. Silva. Teaching nonlinear modeling, simulation, and control of electronic power converters using MATLAB/SIMULINK. *Education, IEEE Transactions on*, 45(3):253–261, aug 2002.

[65] P. Malcovati, S. Brigati, F. Francesconi, F. Maloberti, P. Cusinato, and A. Baschirotto. Behavioral modeling of switched-capacitor sigma-delta modulators. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, 50(3):352–364, mar 2003.

[66] Satish Batchu. Automatic extraction of behavioral models from simulations of analog/mixed-signal (AMS) circuits. Master's thesis, University of Utah, 2011.

[67] Chris Lattner. LLVM. In Amy Brown and Greg Wilson, editors, *The Architecture of Open Source Applications*, volume 1. 2011.

[68] Anthony J Caristi. *IEEE-488, General Purpose Instrumentation Bus Manual.* Academic Press, Inc., 1990.

[69] D. Cheij. A software architecture for building interchangeable test systems. In *AUTOTESTCON Proceedings, 2001. IEEE Systems Readiness Technology Conference*, pages 16 –22, 2001.

[70] A. Pramanick, R. Krishnaswamy, M. Elston, T. Adachi, Harsanjeet Singh, B. Parnas, and L. Chen. Test programming environment in a modular, open architecture test system. In *Test Conference, 2004. Proceedings. ITC 2004. International*, pages 413 – 422, oct. 2004.

[71] Travis E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[72] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' Guide*, volume 9. Society for Industrial Mathematics, 1987.

[73] Jack J Dongarra, Jim R Bunch, GB Moler, and George W Stewart. *LINPACK Users' Guide*. Number 8. Society for Industrial Mathematics, 1987.

[74] Brian T Smith, James M. Boyle, and Jack J Dongarra. Matrix eigensystem routines — EISPACK guide. *Lecture Notes in Computer Science, Berlin: Springer, 1976, 2nd ed.*, 1, 1976.

[75] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Commun. ACM*, 55(2):55–61, February 2012.

[76] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Laboratory (LANL), 2008.

[77] John Ellson, Emden R Gansner, Eleftherios Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz and dynagraph — static and dynamic graph drawing tools. In *Graph Drawing Software*, pages 127–148. Springer, 2004.