

**ALIAS: ABSTRAÇÃO DE CIRCUITOS
ANALÓGICOS PARA VERIFICAÇÃO DE
SISTEMAS DIGITAIS**

ABNER LUÍS PANHO MARCIANO

**ALIAS: ABSTRAÇÃO DE CIRCUITOS
ANALÓGICOS PARA VERIFICAÇÃO DE
SISTEMAS DIGITAIS**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: ANTÔNIO OTÁVIO FERNANDES
COORIENTADOR: CLAUDIONOR JOSÉ NUNES COELHO JÚNIOR

Belo Horizonte

Março de 2015

ABNER LUÍS PANHO MARCIANO

**ALIAS: ANALOG CIRCUIT ABSTRACTIONS FOR
DIGITAL SYSTEMS VERIFICATION**

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: ANTÔNIO OTÁVIO FERNANDES
CO-ADVISOR: CLAUDIONOR JOSÉ NUNES COELHO JÚNIOR

Belo Horizonte

March 2015

© 2015, Abner Luís Panho Marciano.
Todos os direitos reservados.

Marciano, Abner Luís Panho

M319a ALIAS: Abstração de Circuitos Analógicos para
Verificação de Sistemas Digitais / Abner Luís Panho
Marciano. — Belo Horizonte, 2015
xxii, 62 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de
Minas Gerais — Departamento de Ciência da
Computação

Orientador: Antônio Otávio Fernandes

Coorientador: Claudionor José Nunes Coelho Júnior

1. Computação - Teses. 2. Circuitos Integrados -
Teses. 3. Sistemas Eletrônicos - Teses.
4. Reconhecimento de padrões - Teses. I. Orientador.
II. Coorientador. III. Título.

519.6*17(043)




UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

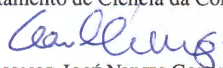
FOLHA DE APROVAÇÃO

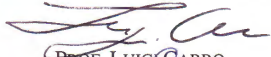
ALIAS: analog circuits abstraction for digital systems verification

ABNER LUIS PANHO MARCIANO

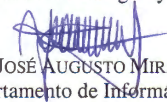
Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. ANTÔNIO OTÁVIO FERNANDES - Orientador
Departamento de Ciência da Computação - UFMG


PROF. CLAUDIONOR JOSÉ NUNES COELHO JÚNIOR - Coorientador
Departamento de Ciência da Computação - UFMG


PROF. LUIGI CARRO
Departamento de Informática - UFRGS


DRA. ANDRÉA IABRUDI TAVARES
Cadence Design Systems


PROF. JOSÉ AUGUSTO MIRANDA NACIF
Departamento de Informática - UFV

Belo Horizonte, 10 de abril de 2015.

Acknowledgments

First, I would like to thank my advisors, Professor Antônio Otávio Fernandes and Professor Claudionor Nunes Coelho. Without their guidance and patience, this dissertation would have been impossible. To Professor Antônio, I would like to thank him for all the support and mentorship throughout my whole academic life. Without him I would never have entered the world of digital hardware. To Professor Claudionor, I thank him for his great teaching, the ideas that drove this work, the discussions, as well as his sharpness, enthusiasm, attention and encouragement.

I'm deeply thankful to all my colleagues and friends at UFMG and Cadence. Among these places I learned with only the best. To Andrea Iabrudi and Professor José Augusto Nacif, my deepest gratitude for their advice, patience and teaching. Furthermore, to my great friends Caio Campos and Tiago Valadares for their support during my time at UFMG.

I cannot overstate how grateful I am to my family and friends. Only they know how much effort I've put into this work. Moreover, for never giving up on me.

Finally, and foremost, I would like to thank Bárbara for always bringing me light and joy. Your care and help when I needed the most will never be forgotten.

I wouldn't have done this if it wasn't for all you. For that, I'm eternally grateful.

“I am always doing that which I cannot do, in order that I may learn how to do it.”

(Pablo Picasso)

Resumo

A crescente demanda por eficiência em área, consumo de energia e desempenho na indústria de circuitos integrados (CI) alinhada à crescente integração em CIs modernos, está tornando a tarefa de verificação cada vez mais complexa e demorada, consumindo até 70% do tempo de desenvolvimento de um CI. Dado que em níveis mais altos de abstração a velocidade de verificação é maior, tal esforço tende a se concentrar nos estágios digitais do fluxo de projeto, onde a maioria das falhas de um CI pode ser encontrada. Contudo, ao se verificar CIs em níveis digitais, componentes analógicos importantes são geralmente ignorados ou vagamente descritos de maneira discreta, comprometendo, assim, a qualidade do processo de validação. Quando tais componentes analógicos são incluídos na análise, eles são integrados no processo de verificação através de ambientes híbridos, os quais são lentos e ineficientes. Sendo assim, dadas as restrições de tempo usuais de um projeto de CI, a validação usando tal abordagem híbrida tende a cobrir menos estados do circuito que uma simulação puramente digital. Dessa forma, visando manter o comportamento de circuitos analógicos no ambiente de verificação, porém, sem o custo de uma verificação híbrida, nós apresentamos uma nova ferramenta, denominada ALIAS (analog logical-intent abstraction synthesizer), bem como uma nova metodologia para a criação de abstrações puramente digitais de circuitos analógicos. Para criar tais abstrações, a metodologia apresentada descreve uma série de passos, desde a aquisição de dados e seu processamento até a geração do modelo digital final. Para isso, abordagens já existentes se valem do uso intensivo de métodos matemáticos e simulações analógicas. Dado que circuitos analógicos a serem integrados em CIs são geralmente verificados de maneira extensiva por profissionais em circuitos analógicos, nós propomos a reutilização de dados provenientes de tais processos. Além de delinear o comportamento de um circuito analógico através de uma perspectiva digital, o ALIAS permite a geração de abstrações com a menor quantidade de intervenção possível por parte do usuário. Como resultado, utilizando vários circuitos analógicos comuns, abstrações sintetizadas com ALIAS demonstraram um ganho considerável em desempenho, bem como um alto nível de precisão. Mais do que isso, as abstrações

criadas com ALIAS escalaram de maneira linear com o crescimento do tamanho da simulação, enquanto simulações analógicas exibiram uma explosão exponencial no tempo de execução.

Palavras-chave: Validação de circuitos, Abstração de circuitos analógicos, Verificação de circuitos.

Abstract

The growing demand for efficiency in area, energy and high-performance in the integrated circuit (IC) industry aligned with the crescent integration of modern ICs is turning the verification process into an even more complex and time-consuming task, taking up to 70% of the IC development time. Provided the faster validation speeds of higher abstraction levels, this effort tends to be concentrated at the digital stages of the design flow, where most IC bugs can be found. Nonetheless, when verifying ICs at digital levels, important analog components are usually left out or poorly described in a discrete way, thus compromising validation process quality. Today, when such analog components are taken into consideration, they are integrated in the verification process through slow and inefficient hybrid verification environments. Therefore, given the usual IC project time constraints, the validation process using such hybrid approach tends to cover fewer circuit states than an entirely digital verification. Therein, aiming to keep analog circuits behavior in the verification environment, but without the cost of a hybrid setting, we present a novel tool, named ALIAS, short for analog logical-intent abstraction synthesizer, along with a new methodology for the creation of purely digital approximations for analog circuits. In order to create such abstractions, the presented methodology describes a series of steps, from data acquisition and processing to the generation of the final digital model. To do so, existing approaches employ intensive mathematical methods and/or analog simulations. Given that analog circuits being integrated into ICs are usually intensively verified by analog experts, we propose reusing simulation data coming from such processes. Aside from outlining the analog circuit behavior from a digital point of view, ALIAS allows the creation of abstractions with as little user intervention as possible. As a result, using various common analog circuits, abstractions synthesized using ALIAS demonstrated significant performance gains as well as a high level of accuracy. Moreover, ALIAS abstractions scaled linearly as simulation sizes grew, while analog simulations displayed an exponential blow up in execution time.

Palavras-chave: Circuit validation, Analog circuit abstraction, Circuit verification.

List of Figures

1.1	IC Validation abstraction levels and speed ¹	2
1.2	A high-level view of ALIAS goal of converting an analog circuit X into a digitized abstraction.	4
2.1	A resistor with a resistance of $R\Omega$	8
2.2	A capacitor with a capacitance of CF	9
2.3	A simple circuit showing branches (the elements), and its nodes (the interconnections), in blue.	10
2.4	A RC circuit (a) and the equivalent circuit after the switch is closed (b).	11
3.1	Kurshan and McMillan [1991] state transition diagram model.	18
3.2	ABCD-NL digital model example for a circuit with 4 DC states.	20
4.1	An example of a typical ALIAS use flow. Dashed lines indicates optional parts of the flow.	21
4.2	System modules data flow.	22
4.3	An example of space/time discretization of an analog trace, resulting on the sequence 0, 0, 1, X, 2 for some net p	23
4.4	An example of sampling analog samples. t_d is the digital time-step and t_a the analog time-step. On the example, $t_d = 4 * t_a$	24
4.5	RC delay line.	24
4.6	RC delay line SPICE simulation trace showing sampled data (a) and detailed sampling sections at 20ns (b) and 40ns (c).	25
4.7	RC delay line samples (a) in the format (d_{in}, d_{out}) and digitized trace (b) from Figure 4.6.	26
4.8	DTDF storage organization scheme. On the figure, $1 \leq k \leq i \leq j \leq n$	28
4.9	A simple trace as list of state tuples.	29
4.10	RC delay line Example 4.3.1.	32

4.11	RC delay line samples (a) in the format (d_{in}, d_{out}) and labeled digitized trace (b) from Example 4.3.1.	32
4.12	Configuration of simple automata ² A_s	34
4.13	Simple automaton model for Example 4.5.5 language L'	34
4.14	Configuration of automaton in parallel ³ A_p	36
4.15	Parallel automaton model for Example 4.5.5 language L'	36
4.16	DOT diagrams for (a) Example 4.6.1 and (b) Example 4.6.2 automaton. Different edges and nodes between representations in red.	40
5.1	RC delay line.	44
5.2	RC repeating L tests speed (a) plain and (b) logarithmic scale charts, demonstrating analog simulations exponential trend and the linear behavior of digital only simulations using ALIAS abstractions (simple and parallel).	46
5.3	RC random tests speed (a) plain and (b) logarithmic scale charts, demon- strating analog simulations exponential trend and the linear behavior of digital only simulations using ALIAS abstractions (simple and parallel).	47
5.4	Analog integrator circuit.	48
5.5	Integrator SPICE simple simulation trace.	48
5.6	Integrator SPICE simple simulation trace showing sampled data.	49
5.7	Integrator repeating L tests speed (a) plain and (b) logarithmic scale charts, demonstrating analog simulations exponential trend and the linear behavior of digital only simulations using ALIAS abstractions (simple and parallel).	50
5.8	Integrator random tests comparing ALIAS abstractions (simple and paral- lel) quality with the amount of scenarios used to create the abstractions for simulations of $1000L$ sampling points.	51
5.9	Typical analog comparator topology.	52
5.10	Typical PLL-based synchronizer topology.	53
5.11	DAC and ADC (a) typical topology and (b) topology used for tests.	54

List of Tables

3.1	Comparison of AMS abstraction approaches	20
5.1	RC repeating L tests speed comparison between an analog simulation and digital only simulation using ALIAS abstractions (simple and parallel). . .	45
5.2	Quality ratios for RC repeating L tests over ALIAS abstractions (simple and parallel).	46
5.3	RC random tests speed comparison between an analog simulation and digital only simulation using ALIAS abstractions (simple and parallel).	47
5.4	Quality ratios for RC random tests over ALIAS abstractions (simple and parallel).	48
5.5	Integrator repeating L tests speed comparison between an analog simulation and digital only simulation using ALIAS abstractions (simple and parallel).	50
5.6	Integrator random tests comparing ALIAS abstractions (simple and parallel) quality with the amount of scenarios used to create the abstractions for simulations of $1000L$ sampling points.	52
5.7	PLL synchronizer basic 10^3 periods simulations speed comparison between an analog simulation and digital only simulation using ALIAS abstractions (simple and parallel).	54
5.8	Quality ratios for PLL synchronizer 10^3 periods simulation tests over ALIAS abstractions (simple and parallel).	54

Contents

Acknowledgments	ix
Resumo	xiii
Abstract	xv
List of Figures	xvii
List of Tables	xix
1 Introduction	1
1.1 Goals	3
1.2 Contributions	4
1.3 Text Organization	5
2 Background	7
2.1 Electronic circuits	7
2.1.1 Basic concepts	8
2.1.2 Circuit elements	8
2.1.3 Circuit networks	9
2.1.4 Circuit dynamics	10
2.2 Discrete Math Concepts	13
2.2.1 Fundamentals	13
2.2.2 Formal languages	15
3 Related Work	17
4 System Architecture	21
4.1 System Overview	21
4.2 A walkthrough of ALIAS	22

4.3	Frontend: Analog Waveform Trace Processor (AWTP)	23
4.4	Discrete Trace Database (DTD)	25
4.5	Synthesis Engine	28
4.6	Backend: Model compiler	33
4.6.1	Automaton models	33
4.6.2	DOT diagrams compilation	38
4.6.3	Verilog models compilation	38
5	Case studies	43
5.1	RC delay line	44
5.2	Integrator	47
5.3	Analog comparator	51
5.4	PLL	52
5.5	DAC and ADC converters	53
6	Conclusions	57
6.1	Contributions	57
6.2	Limitations and Future Work	58
6.2.1	Automata history and transition improvements mechanisms	58
6.2.2	Segment Boundary Inference	58
6.2.3	Recognition of Common Data Patterns	58
6.2.4	Data and control signals handling	58
6.2.5	Formal verification	58
	Bibliography	59

Chapter 1

Introduction

The growing demand for efficiency in area, energy and high-performance on the integrated circuit (IC) industry lead to the integration of several circuits in fewer or, ideally, a single chip [Jaeger, 1987]. Following this trend, real modern application ICs are usually composed by several analog and digital sub-circuits, for what they are denominated mixed-signal systems (MSS).

As the circuit complexity escalates, IC verification is getting more and more complex, and can take up to 70% of the development time [Barke et al., 2009]. The main established techniques for verifying circuits are divided among emulation, formal verification and simulation based methods, from which simulation is still the main motor and standard in IC verification [Stroud et al., 2009].

As depicted by Figure 1.1, IC verification can be performed at several levels of abstraction, ranging from high-level models to low-level models. Taking simulation as basis, the more abstract, the faster the circuit under test can be verified, as shown in Figure 1.1. On this line of thinking, although less precise, as most bugs can be found at higher abstraction levels [Kularatna and Kyung, 2008], verification tends to be more concentrated at the digital stages of the design flow, going from higher abstraction levels to lower ones.

Validation of digital circuits has several advantages over analog or mixed-signal (AMS) circuits, such as faster verification environments, including powerful ASIC emulation and prototyping platforms, and, more importantly, all advantages of the discrete domain. Aligned to highly automated digital design flows, and the less sensitivity to electrical noise, modern ICs are composed by from 50% to 90% of digital blocks [Chen et al., 2012]. Taking up to half of a modern IC, AMS sub-circuits are one of the most problematic blocks [Liberati et al., 2002], which justifies the actively growing literature

¹Based on [Rashinkar et al., 2000].

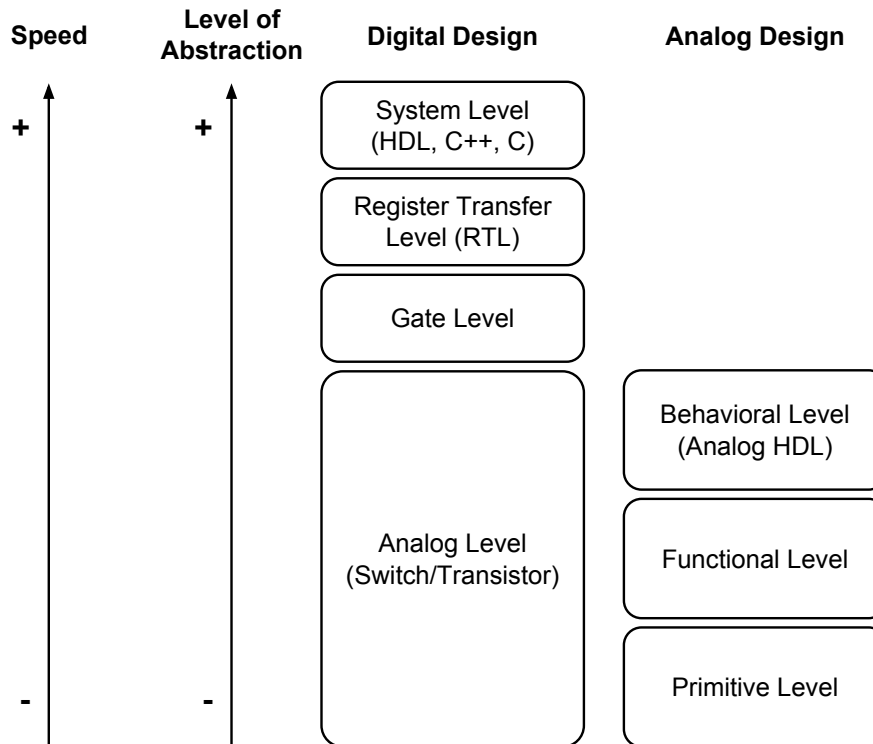


Figure 1.1: IC Validation abstraction levels and speed¹.

[Karthik et al., 2014; Karthik and Roychowdhury, 2013; Aadithya and Roychowdhury, 2012; Steinhorst and Hedrich, 2008; Hartong et al., 2002; Asarin et al., 2001; Silva and Krogh, 2000; Henzinger et al., 1997; Kurshan and McMillan, 1991] on the field. However, little has been done to aid digital circuit engineers on verifying the integration of their digital blocks with AMS ones. The problem deepens with common factors on the IC industry, ranging from enormous AMS netlists to poor block descriptions. It gets even more problematic as usually both analog and digital parts are intensively verified on completely different abstraction levels.

As verification is focused on higher levels of abstraction for digital circuits, to verify such digital components, the interface of AMS circuits is treated as digital. For that, either slow and complex hybrid simulation techniques [Ghasemi and Navabi, 2005; Rashinkar et al., 2000; El Tahawy et al., 1993], which simulates both digital and analog circuits at the same time, or digital simplifications of the AMS sub-circuit are employed. As digital simplifications tends to be far faster than also simulating an analog circuit, to create such simplification one should analyze the analog circuit. As digital engineers are usually specialized in the digital domain, most have few or no knowledge in analog design, thus, it is common to have poor simplifications, omitting several details that should have been taken into account in a digital environment [Semiconductor Research

Corporation, 2008].

Most digital circuits are synchronous, that is, they depend on periodic *clock* signals to coordinate its actions. Nevertheless, digital circuits still need to synchronize with the outside world, which is inherently analog. To perform this bridge between digital portions of an IC and the outside world, analog circuits are employed. Analog circuits are dynamic systems that update its outputs upon changes on its input signals and internal components behavior. Therein, the core idea of this work is to create synchronous digital simplifications of analog circuits, allowing such abstractions to be paired with any digital interfacing block.

Instead of creating models directly from differential equations, as in [Kurshan and McMillan, 1991; Hartong et al., 2002; Steinhorst and Hedrich, 2008; Karthik and Roychowdhury, 2013], we propose a simulation based approach, as done by [Aadithya and Roychowdhury, 2012; Karthik et al., 2014]. Moreover, as analog blocks are intensively verified at an analog level [Rashinkar et al., 2000], an analog engineer perform several tests, being analog simulations one of the main techniques and products of that work. Thus, as a circuit’s most relevant behaviors are usually exercised during analog verification and aiming to reuse these results, differently from [Aadithya and Roychowdhury, 2012; Karthik et al., 2014], and to avoid extra computational time, we propose the creation of such abstractions based on preexisting analog simulations.

1.1 Goals

With this work we aim to create a new methodology for verifying digital circuits in the presence of the nearest approximation possible to an analog circuit, targeting performance as well as a reasonable level of accuracy. In essence, this methodology consists on a series of steps to bring the analog behavior of a circuit into the digital world, which can be used from a system-level model to the gate-level phase of the IC development flow.

Using the proposed methodology, analog circuits interfacing with a digital circuit under test are replaced by automated digital simplifications created with as little as possible user intervention. To further enable this project, we design and implement a system, called ALIAS, short for **analog logical-intent abstraction synthesizer**, that, as illustrated by Figure 1.2, allow the creation of such abstractions and, on the other hand, also enables the user to understand the underlying analog circuit behavior, from a digital perspective. Finally, we target to verify the proposed solution against the metrics of speed, complexity and quality.

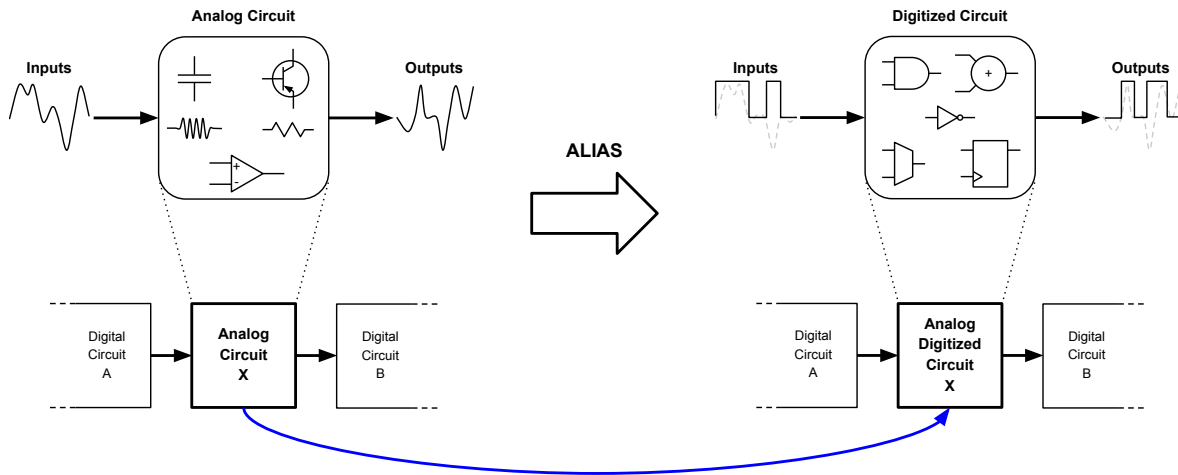


Figure 1.2: A high-level view of ALIAS goal of converting an analog circuit X into a digitized abstraction.

To complement that, we pursue the following secondary objectives:

- Implement an algorithm to extract an automata that mimics the behavior of an analog system based on its observed behavior;
- Convert the created automata into HDL language models;
- Create a testbench of common analog designs and possible digital counter-parts.

1.2 Contributions

With this work we add a powerful methodology and tool to aid digital engineers on testing digital systems along with analog ones. By outlining an analog circuit behavior from a digital perspective, digital engineers can better understand a block and focus their tests. Also, this will further enable digital engineers on finding bugs earlier in the IC development flow, reducing the costs of the project and time-to-market.

Aside from demanding little user intervention, ALIAS doesn't require running any new analog simulations by reusing already existing tests performed by analog engineers. More importantly, the abstractions synthesized presented significant performance gains as well as a reasonable level of accuracy on several scenarios and circuit types, specially when compared to an analog simulator.

1.3 Text Organization

This dissertation is organized as follows: Chapter 2 presents a brief description of the theory behind analog circuits as well as key discrete math concepts used throughout this work.

Chapter 3 presents the state-of-art of AMS circuit abstraction and validation, summarizing their differences against this work's proposed approach.

Chapter 4 starts by showing an overview of the proposed methodology as a system. Next, each module composing the system is detailed and exemplified.

Then, experimental results are presented on Chapter 5 comparing speed, complexity and quality of the produced models for several circuit types.

Finally, conclusions, limitations and future work are discussed and detailed at Chapter 6.

Chapter 2

Background

2.1 Electronic circuits

As defined by Alexander and Sadiku [2008], an electric circuit is an interconnection of electrical elements through conducting wires. These elements includes capacitors, inductors, resistors, voltage supplies, current supplies, transistors, diodes and so on.

To understand the electrical phenomena, we start with some basic concepts, including charge, current and voltage. We then introduce some basic circuit elements as well as further laws and concepts required to understand the analyzed circuits' dynamics.

For the sake of brevity, this chapter aims to cover just the required theory to understand examples used throughout this text. Thus, we only focus on electrical circuits that can be directly modeled into linear time-invariant (LTI) systems¹, as defined in Definition 2.1.1. The interested reader should refer to [Horowitz and Hill, 2006], as a electronic design reference, [Alexander and Sadiku, 2008] for circuit analysis and dynamics, and [Friedland, 2012] for linear and time-invariant systems modeling.

Definition 2.1.1. Let x be an electric circuit signal and its value over time t , defined by a function $x(t)$. Therein, a system is then classified as *linear* if given any two input signals $x_1(t)$ and $x_2(t)$ producing, respectively, outputs $y_1(t)$ and $y_2(t)$, for any constant γ then the input signal $\gamma x_1(t) + \gamma x_2(t)$ generates $\gamma y_1(t) + \gamma y_2(t)$. A system is said to be *time-invariant* if for any $\delta \in \mathbb{R}$, the time-shifted input $x(t - \delta)$ produces the output $y(t - \delta)$. A system that is both linear and time-invariant is denominated as a linear time-invariant (LTI) system.

¹Elements such as capacitors, inductors, resistors, voltage and current supplies are LTI, and circuits formed with just these elements are LTI systems as well. However, components like transistors and diodes are non-LTI, as systems using such elements.

2.1.1 Basic concepts

The electric charge is one of the most basic properties of matter. In a simple way, the electric charge measures in a quantized way the extent to which an object has more or fewer protons than electrons. Measured by the International System of Units (SI) in Coulombs (C), an electron, is known to have a negative charge of $e = -1.602 * 10^{-19}$, and a proton, a positive charge to the same amount as the electron of $p = 1.602 * 10^{-19}$.

The flow of electric charges in an object, or the charge change rate over time, is known as electric current. The relation between the electric current i , measured in amperes (A) in the SI, charge q and time t (in seconds), can be expressed as:

$$i = \frac{dq}{dt} \quad (2.1)$$

Thus, it gets evident that:

$$1A = 1 \frac{C}{s}$$

However, for an electron to move it needs energy, in the form of work. The work w , measured in Joules (J), done per unit of charge to move it from a point a to a point b is known as voltage. Measured in volts (V) on the SI, it can be mathematically expressed as:

$$v_{ab} = \frac{dw}{dq} \quad (2.2)$$

It is also clear that:

$$1V = 1 \frac{J}{C}$$

2.1.2 Circuit elements

There are two types of elements in an electric circuit. The first, known as active elements, are capable of generating energy, such as a battery. On the other hand, passive elements, like resistors, capacitors and inductors, are not.



Figure 2.1: A resistor with a resistance of $R\Omega$.

Resistors, as shown in 2.1, are elements able to resist the flow of electric current. Per Ohm's law [Alexander and Sadiku, 2008], in the presence of such elements, the

voltage v is directly proportional to the electric current flow i , being the constant of proportionality, the resistance, R , measured in ohms (Ω). Mathematically speaking:

$$v = iR \Rightarrow R = \frac{v}{i} \quad (2.3)$$

Thereafter,

$$1\Omega = 1 \frac{V}{A}$$

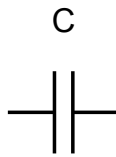


Figure 2.2: A capacitor with a capacitance of CF .

A capacitor, shown in Figure 2.2, according to Alexander and Sadiku [2008], is a passive element designed to store energy using an electric field. For that, two conducting plates separated by an insulator. When a voltage v is applied to the capacitor plates, a positive charge q is deposited in one place and $-q$ in the other plate. The amount of charge stored q is directly proportional to the applied voltage v , being the constant of proportionality, the capacitance, C , measured in farads (F). Thus, we have:

$$q = Cv \quad (2.4)$$

Thereafter,

$$1F = 1 \frac{C}{V}$$

Differentiating both sides of Equation 2.4 with regard to t , we have:

$$\frac{dq}{dt} = C \frac{dv}{dt} \Rightarrow i = C \frac{dv}{dt} \quad (2.5)$$

2.1.3 Circuit networks

As introduced circuit is an interconnection of elements, forming networks. An electrical element, such as a resistor or voltage supply, defines a *branch*, while the connections between branches, defines a *node*.

Although key, Ohm's law it not sufficient to work on complex circuit network topologies. Thus, two laws play an important role when analyzing such networks: Kirchhoff's current law (KCL) and Kirchhoff's voltage law (KVL).

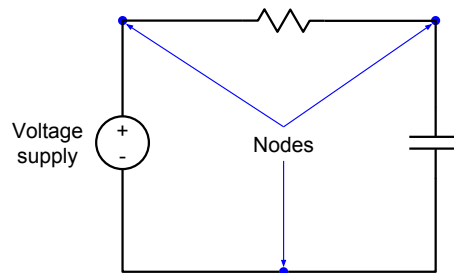


Figure 2.3: A simple circuit showing branches (the elements), and its nodes (the inter-connections), in blue.

KCL bases on the charge conservation on a electrical system, and states that the algebraic sum of currents I entering (or leaving) a node is zero. Thus, given the number M of branches and that i_m is the m th current reaching that node, we have:

$$I = \sum_{m=1}^M i_m = 0 \quad (2.6)$$

The last law, KVL, states that given a path in a circuit that starts at a node and ends up on the same node without repeating any node in the path, the algebraic sum V of all voltages in that path is zero. With v_n the n th voltage from in the path, N the amount of branches in the path, we have:

$$V = \sum_{n=1}^N v_n = 0 \quad (2.7)$$

2.1.4 Circuit dynamics

Over-time, it is possible to describe a circuit's behavior per its current or voltage, depending on the analysis point of view. For that end, the system elements interconnection points, known as nodes, are taken as basis and its behavior describing equations combined, forming a system of differential equations, which usually can be converted to a system of simpler differential equations, known as ordinary differential equations (ODEs).

Example 2.1.1. In the circuit example from Figure 2.3, a switch is closed at $t = 0$, after which the circuit capacitor starts charging.

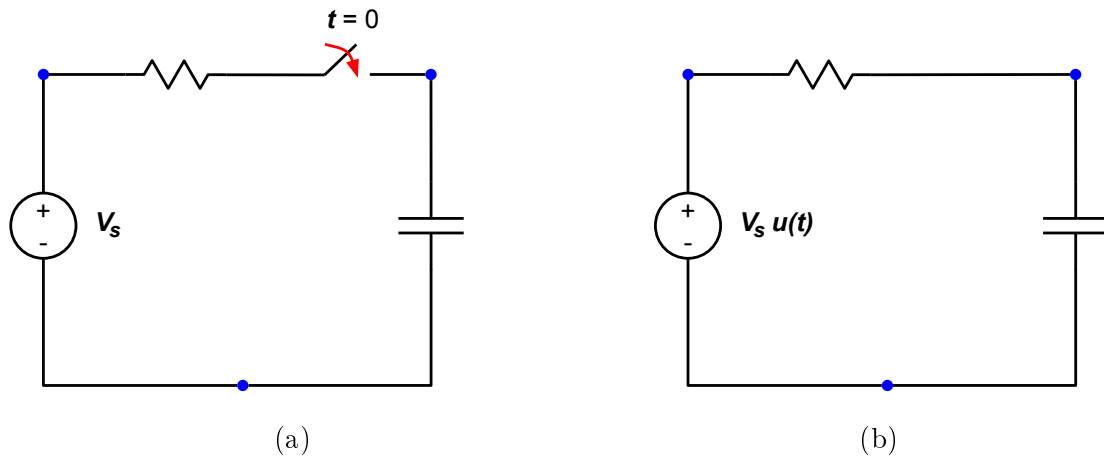


Figure 2.4: A RC circuit (a) and the equivalent circuit after the switch is closed (b).

The switch behavior can be modeled with the step function²:

$$u(t) = \begin{cases} 0 & \text{if } t < 0 \\ 1 & \text{if } t > 0 \end{cases} \quad (2.8)$$

Assuming an initial voltage V_0 on the capacitor, since the capacitor cannot charge immediately, right after the switch is closed the voltage on the capacitor will still be V_0 . Per KCL, the sum of currents leaving the system top node is 0, therefore:

$$i_C + i_R = 0 \quad (2.9)$$

Thus,

$$C \frac{dv}{dt} + \frac{v - V_s u(t)}{R} = 0$$

$$\frac{dv}{dt} + \frac{v}{RC} = \frac{V_s}{RC} u(t) \quad (2.10)$$

²For more on step functions, refer to Alexander and Sadiku [2008].

With v being the voltage across the capacitor. For $t > 0$ we have:

$$\begin{aligned}\frac{dv}{dt} + \frac{v}{RC} &= \frac{V_s}{RC} \\ \frac{dv}{dt} &= -\frac{v - V_s}{RC} \\ \frac{dv}{v - V_s} &= -\frac{dt}{RC}\end{aligned}\tag{2.11}$$

Integrating on both sides,

$$\begin{aligned}\ln(v - V_s)\Big|_{V_0}^{v(t)} &= -\frac{t}{RC}\Big|_0^t \\ \ln(v - V_s) - \ln(V_0 - V_s) &= -\frac{t}{RC} + 0 \\ \ln\frac{v - V_s}{V_0 - V_s} &= -\frac{t}{RC}\end{aligned}\tag{2.12}$$

Taking the exponential on both sides,

$$\begin{aligned}\frac{v - V_s}{V_0 - V_s} &= e^{-\frac{t}{RC}} \\ v - V_s &= (V_0 - V_s)e^{-\frac{t}{RC}}\end{aligned}\tag{2.13}$$

And, as by v we mean $v(t)$,

$$v(t) = V_s + (V_0 - V_s)e^{-\frac{t}{RC}}, \quad t > 0\tag{2.14}$$

Finally,

$$v(t) = \begin{cases} V_0 & \text{if } t < 0 \\ V_s + (V_0 - V_s)e^{-\frac{t}{RC}} & \text{if } t > 0 \end{cases}\tag{2.15}$$

This equation, known as a first-order differential equation, a form of an ordinary differential equation (ODE), describes the circuit behavior with regard to the voltage change rate over time.

2.2 Discrete Math Concepts

The sole purpose of this section is to introduce the reader to some basic discrete math concepts, from the definition of an element to collections of elements and operations and properties of these collections. Finally, using the concepts already presented, we define a formal language.

2.2.1 Fundamentals

Let us start with a standard definition of an element, which can be understood as an arbitrary unit of data, such as numbers, words, texts, and images.

Example 2.2.1. The word “*earth*”, the number 7, the phrase “*this is a phrase*”, all numbers in \mathbb{C} , the formula “ $e = mc^2$ ”, the random arrangement of characters “*Barbara Marques*” are all valid examples of elements.

To represent a collection of elements, we define its elementar containers — *tuple*, *list* and *set* — and the operations to be used over them.

Definition 2.2.1. Let $T = (e_1, \dots, e_n)$ denote a tuple, an immutable finite group composed by n elements, which, for any two tuples $A = (a_1, \dots, a_n)$ and $B = (b_1, \dots, b_m)$:

- A_i refers to the i -th element from A , that is $A_i = a_i$;
- $|A|$ denote the number elements in A ;
- the order that elements are layed out matters, that is, a tuple A is equal to a tuple B , as long as $|A| = |B|$ and $A_i = B_i, 1 \leq i \leq |A|$.

Example 2.2.2. Given the tuples $A = (1, \text{“abc”}, 3.0)$, $B = (3.0, \text{“abc”}, 1)$, $C = (1, \text{“abc”}, 3.0)$ and $D = ((3.0, \text{“abc”}, 1), 1)$, we have:

- $|A| = 3$, $|B| = 3$, $|C| = 3$ and $|D| = 2$.
- $A_1 = 1$, $B_2 = \text{“abc”}$, $D_1 = (3.0, \text{“abc”}, 1)$;
- As D_1 is also a tuple, $|D_1| = 3$;
- $A = C$ and $B = D_1$
- $A \neq B$, $A \neq D$, $B \neq C$, $B \neq D$ and $C \neq D$;
- For $E = D_1$, $E_1 = 3.0$.

Definition 2.2.2. Let $L = [e_1, \dots, e_n]$ denote a list, a mutable finite sequence of n elements. For any two lists $A = [a_1, \dots, a_n]$ and $B = [b_1, \dots, b_m]$:

- A_i refers to the i -th element from A , that is $A_i = a_i$;
- $|A|$ denote the number elements in A ;
- the order elements are layed out matters, that is, a list A is equal to a list B , as long as $|A| = |B|$ and $A_i = B_i$, $1 \leq i \leq |A|$;
- the concatenation of A and B as $C = AB = [A_1, \dots, A_n, B_1, \dots, B_m]$;

Example 2.2.3. Given the lists $A = [1, ("abc", 3.0), 2i]$, $B = [2i, ("abc", 3.0), 1]$, $C = [1, 2i, ("abc", 3.0)]$ and $D = [[(2i, "abc", 3.0), 1], 1]$, we have:

- $|A| = 2$, $|B| = 2$, $|C| = 2$ and $|D| = 2$;
- $A_1 = 1$, $B_1 = 2i$, $C_3 = ("abc", 3.0)$ and $D_1 = [(2i, "abc", 3.0), 1]$;
- As D_1 is also a list, $|D_1| = 2$;
- $A \neq B$, $A = C$, $A \neq D$, $A \neq D_1$, $B \neq C$, $B \neq D$, $B = D_1$, $C \neq D$ and $C \neq D_1$;
- For $E = D_1$, $E_1 = (2i, "abc", 3.0)$;
- $AC = [1, ("abc", 3.0), 2i, 1, 2i, ("abc", 3.0)]$, $CD_1 = [1, 2i, ("abc", 3.0), 2i, ("abc", 3.0), 1]$.

Definition 2.2.3. Let $L = \{e_1, \dots, e_n\}$ denote a set, a mutable unordered finite group of n elements, which, for any two sets $A = \{a_1, \dots, a_n\}$ and $B = \{b_1, \dots, b_m\}$:

- $|A|$ denote the number elements in A ;
- the order elements are layed out does **not** matter, that is, a set A is equal to a set B , as long as both sets contains the same elements, that is, $A \cap B = A = B$;
- a set A can only equal to a set B if $|A| = |B|$ and $\forall a \in A, a \in B$ and $\forall b \in B, b \in A$, meaning that the order is not relevant when comparing two sets;
- $C = A \cap B = \{e \mid e \in A \text{ and } e \in B\}$;
- $C = A \cup B = \{e \mid e \in A \text{ or } e \in B\}$.

Example 2.2.4. Given the sets $A = \{(1.0, -1.0), ["abc", 3.0], 2i\}$, $B = \{2i, ["abc", 3.0], (1.0, -1.0)\}$, $C = \{(1.0, -1.0), 2i, ["abc", 3.0]\}$ and $D = \{["abc", 3.0], (1.0, -1.0), 2i, 1\}$, we have:

- $|A| = 3$, $|B| = 3$, $|C| = 3$ and $|D| = 2$;

- $A = B, A = C, A \neq D, B = C, B \neq D$ and $C \neq D$;
- $A \cup C = \{(1.0, -1.0), ["abc", 3.0], 2i\}$,
 $C \cup D = \{2i, ["abc", 3.0], (1.0, -1.0), \{("abc", 3.0), (1.0, -1.0), 2i\}\}$;
- $A \cap C = \{(1.0, -1.0), ["abc", 3.0], 2i\}$,
 $C \cap D = \{\}$.

2.2.2 Formal languages

We define a language as follows:

Definition 2.2.4. Let $L = (\Sigma, \lambda, \Gamma, \Psi)$ be a standard definition of a language and its algebra:

- A symbol is an arbitrary item, named after a glyph, such as a letter or a number;
- An alphabet Σ is a non-empty collection of symbols;
- A string is an arbitrary finite concatenation of symbols from Σ ;
- A language L is a set of strings over an alphabet Σ ;
- The empty string, denoted by λ , is a string containing zero symbols ($|\lambda| = 0$);
- The set of all strings over Σ is Σ^* , thus, for any language L over Σ , $L \subseteq \Sigma^*$;
- Γ is the set of string operators, which, for any two strings x and y , is:
 - The concatenation of two strings $x = x_1 \cdots x_m$ and $y = y_1 \cdots y_n$ is defined by $xy = x_1 \cdots x_m y_1 \cdots y_n$;
 - The repetition of a string x for n times as $w = x^n = \overbrace{xx \cdots xx}^{n \text{ times}}$, with $x^0 = \lambda$.
- Ψ the set of language operators, which, for any two languages L_1 over Σ_1 and L_2 over Σ_2 , is:
 - $L = L_1 \cup L_2$ a language over $\Sigma_1 \cup \Sigma_2$ composed by strings in L_1 or L_2 ;
 - $L = L_1 \cap L_2$ a language over $\Sigma_1 \cap \Sigma_2$ composed by strings in L_1 and L_2 ;
 - $L = L_1 - L_2$ a language over Σ_1 composed by strings in L_1 and not in L_2 ;
 - The concatenation of two languages, as $L = L_1 L_2 = \{xy \mid x \in L_1, y \in L_2\}$;
 - The consecutive concatenation of a language L for n times as $L^n = \overbrace{LL \cdots LL}^{n \text{ times}}$, with $L^0 = \{\lambda\}$;

- The *Kleene closure* over L as $L^* = \bigcup_{n \in \mathbb{N}} L^n$;
- The *positive Kleene closure* over L as $L^+ = L^* - \{\lambda\}$.

Example 2.2.5. After Definition 2.2.4, let the symbols a, b, c, d, e, f, g, h form the alphabets $\Sigma_1 = \{a, b, e, f\}$ and $\Sigma_2 = \{c, d, e, g, h\}$. Thus, aba, fae, bf and ee are strings over Σ_1 and $cd, chdg, ee$ and λ strings over Σ_2 . From these strings, $L_1 = \{aba, fae, bf, ee\}$ is a language over Σ_1 and $L_2 = \{cd, chdg, ee, \lambda\}$ a language over Σ_2 .

Thus, for the language operations Ψ we would have:

- $L_3 = L_1 \cup L_2 = \{aba, fae, bf, cd, chdg, ee, \lambda\}$ over $\Sigma_3 = \Sigma_1 \cup \Sigma_2 = \{a, b, c, d, e, f, g, h\}$;
- $L_4 = L_1 \cap L_2 = \{ee\}$ over $\Sigma_4 = \Sigma_1 \cap \Sigma_2 = \{e\}$;
- $L_5 = L_1 - L_2 = \{aba, fae, bf\}$ over Σ_1 , and $L_6 = L_2 - L_1 = \{cd, chdg, \lambda\}$ over Σ_2 ;
- $L_7 = L_4 L_5 = \{eeaba, eefae, eebf\}$ over $\Sigma_5 = \{a, b, e, f\}$, and $L_8 = L_6 L_4 = \{cdee, chdgee, ee\}$ over $\Sigma_6 = \{c, d, e, g, h\}$;
- $L_9 = L_4^0 = \{\lambda\}$,
 $L_{10} = L_4^1 = \{ee\}$,
 $L_{11} = L_4^2 = \{eeee\}$
 $L_{12} = L_4^3 = \{eeeeee\}$
 $L_{13} = L_4^4 = \{eeeeeeee\}$
and $L_4^5 = \{eeeeeeeeee\}$, all over Σ_2 ;
- $L_{14} = L_4^* = \{\lambda, ee, eeee, eeeee, eeeeeee, \dots\}$ over Σ_2 ;
- $L_{15} = L_4^+ = \{ee, eeee, eeeee, eeeeeee, \dots\}$ over Σ_2 .

Chapter 3

Related Work

Dominated by digital logic, the increasing complexity of modern ICs fostered the creation of new and faster methods on verifying a circuit, including, but not limited to, formal verification methods, equivalence checking, and emulation and co-simulation environments [Stroud et al., 2009; Rashinkar et al., 2000]. Aside from enhancing the verification environment itself, automatic ways on detecting and synthesizing high-level models for digital circuits based on pre-existing simulation data have been proposed. *Inferno* [Isaksen and Bertacco, 2006] generates high-level transition models and SystemVerilog Assertions (SVA) [IEEE, 2013] properties by detecting transactions on digital simulation waveforms. Also using digital simulation data, newer tools, like *Gold-Mine* [Vasudevan et al., 2010], employs data mining and pattern matching algorithms to synthesize SVA properties.

In spite of all new methods and improvements in the analog world, the speed gap between analog and digital verification is still significant, driving the research for new ways on creating high-level models of analog circuits.

Kurshan and McMillan [1991] proposed a semi-algorithmic way of constructing high-level state machines that preserves the analog behavior of an analog circuit. For that, the system is initially modeled after a network of voltage-variant current and voltage sources and capacitances. The circuit state is a vector of voltages $\mathbf{x} = \{x_0, \dots, x_n\}$ describing the voltage of each node in the network, with the exception of circuit inputs and nodes controlled by independent voltage sources, which are grouped in a vector $\mathbf{y} = \{y_1, \dots, y_m\}$. The system dynamics over time (t) is then modelled after ordinary differential equations (ODEs). As all systems taken into consideration by the author only handles currents in each element of the system that are either monotonic non-increasing or monotonic non-decreasing, given upper and lower bound values for \mathbf{y} elements, one can calculate upper and lower bound values for \mathbf{x} by solving the sys-

tem ODEs. Given a $\Delta(t)$ small enough to capture the system dynamics and value bounds/ranges for mapping $x_i \in \mathbf{x}$ into an alphabet $\Sigma = \{0, 1, X\}$, with X representing an error/illegal state, it is possible to create a discrete automata that describes the circuit behavior, as depicted by Figure 3.1, in the form of a n -dimension space state transition diagram. With this model, the high-level system can be verified against a set of properties, which are encoded as automatons that recognizes the analog discrete model alphabet Σ .

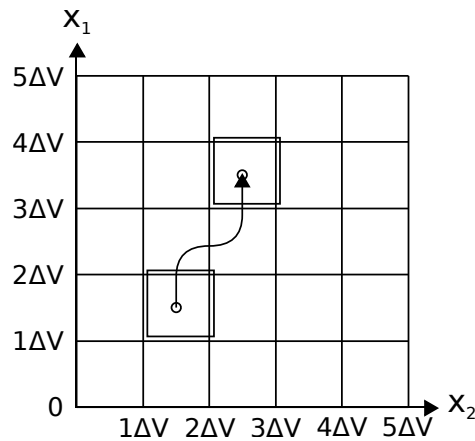


Figure 3.1: Kurshan and McMillan [1991] state transition diagram model.

Although promising, this method has a limitation on the circuit elements electrical current behavior, which limits the amount of circuits that the system can support. Also, it demands a high level of user intervention, as one needs to set and define the bounds of all x and y elements, as well as solving the system ODEs.

On a different approach from Kurshan and McMillan [1991] and similar approaches [Steinhorst and Hedrich, 2008; Hartong et al., 2002], Aadithya and Roychowdhury [2012] introduced DAE2FSM, an incremental learning technique, based on the L^* algorithm [Angluin, 1987], to create FSMs for analog circuits based on the execution of SPICE [Nagel and Pederson, 1973] simulations. The algorithm is composed by two main modules, the *learner* and the *teacher*, both configured with the same input and output alphabets. The learner module is purely discrete, while the teacher has access to the analog circuit SPICE netlist, a SPICE simulator and rules to convert from analog sequences to output symbols, and from input symbols to analog sequences. The learner holds a table T describing system transitions based on the input/output alphabets. To populate it, the learner asks the teacher for the output symbols for a given input sequence. As T is populated by the learner, T is verified for inconsistencies. Once an inconsistency is found, the table grows multiplicatively in regard to the input alphabet size, and the remaining new entries get populated, when T will be checked

again for inconsistencies. If no inconsistencies are found, it will check with the teacher if T is correct, which then execute SPICE simulations to verify it. If T is not sane, T grows proportionally to the number of wrong entries. Aside from the fact that the teacher is a SPICE simulation intensive module, the main downside of this technique is that it is exponential in size and space, as at each inconsistency found, T grows at a multiplicative rate.

To avoid DAE2FSM’s explicit state-enumeration, Karthik and Roychowdhury [2013] presented a tool, named ABCD-L, to create digital models of linear-only analog circuits based on the circuit netlist alone. For that, the system is specified or converted to an ODE, which is then converted to a form where it can be solved analytically. The key idea behind this technique is to express the analytical solution to such equations through digital logic. The first composing elements of these equations, the state of the circuit nodes, are modeled into selectable precision (in number of bits) logic units (LUs), which contains only counters and registers to evaluate its correspondent ODE equation in its analytical form. The system then combines the LU’s results in a purely combinational manner, yielding in the circuit outputs.

While ABCD-L handles well linear systems, ignoring the intensive use of mathematical methods to model the system as an ODE and convert the ODEs into boolean models, ABCD-L still lacked support to non-linear systems, which is the case of many common AMS circuits. To fill this gap, Karthik et al. [2014] introduced ABCD-NL, which has all the benefits of the digital models of ABCD-L and can handle several non-linear systems. As in ABCD-L, ABCD-NL digital model represents the circuit’s input and outputs using a finite number of bits. The model core states, named *DC states*, are associated with all circuit inputs discretized combinations. To capture the analog circuit transient behavior, the system executes SPICE simulations for each pair of DC states, which are then analyzed to create additional states, called *transient states*, that model the circuit transient behavior, as illustrated by Figure 3.2. The number of such states will depend on how long the model takes to transition from one DC point to another and in the number of non-input and non-output signals taken into account by the system. After this base model is created, all state transitions related to DC nodes are defined, and the system can now determine the remaining arcs for transient states based on a interpolation-based heuristic. At mere $2x$ speedups against a high-performance industrial analog simulator, the generated model performance is still far from digital simulation performance, as it is exponential in size with regard to the number of inputs and outputs of the system.

Given an arbitrary circuit, with x being the set of circuit element states, y the set of circuit inputs, Table 3.1 presents a comparison between the presented art and

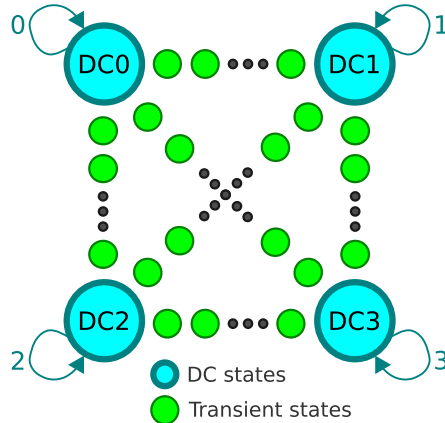


Figure 3.2: ABCD-NL digital model example for a circuit with 4 DC states.

Approach	Model Size	Circuit compatibility	Synthesis Method
Kurshan and McMillan [1991]	$O(\Delta V^{ \mathbf{x} })$	Linear and non-linear systems ¹	Mathematical
DAE2FSM	Exponential	Digitized analog ²	On-demand SPICE simulations
ABCD-L	$O(\Delta V * \mathbf{x})$	Linear systems	Mathematical
ABCD-NL	$\Omega(\Delta V^{ \mathbf{y} })$	Linear and non-linear systems	On-demand SPICE simulations
ALIAS	$O(K * \Delta V * \mathbf{y})$, constant ³ K	Linear and non-linear systems	Pre-existing SPICE simulations

¹ Monotonic increasing/decreasing only ODEs.² Tested only against analog circuits with near digital behavior.³ K is the largest string size from an automata, as detailed on Chapter 4.

Table 3.1: Comparison of AMS abstraction approaches

this work’s proposed approach. As summarized by Table 3.1, although much research has been done on the field, most current approaches still lack on either performance or circuit compatibility. More, little concern has been directed towards the model synthesis speed, relying on heavy mathematical computations or intensive SPICE simulations. **Our approach addresses these problems, as from the premise that the analog circuit designer employed a reasonable amount of effort during verification, we can then assume that the existing simulation data already covers a portion of circuit states that allows the creation of a high quality approximation**, thus providing a compatible, fast, convenient and yet accurate methodology.

Chapter 4

System Architecture

4.1 System Overview

As previously exposed, we propose a SPICE simulation based approach for building digital simplifications of analog circuits. For that, we aiming to avoid extra computational time, we reuse SPICE simulation data produced during the verification phase of the targeted analog circuit.

The core idea of the system on creating abstractions comes from the observation that some electronic systems usually start at an idle state, until they are signaled to perform an action, to then perform it and settle back in an idle state. On digital circuits this is usually the case for transaction-based systems, as shown in [Isaksen and Bertacco, 2006], which presents a way of detecting transactions based on this characteristic. This same pattern can also be observed in several analog circuits used in ICs, from delay chains to PLLs. We thus map Isaksen and Bertacco [2006] transaction extraction ideas as a way of capturing and reproducing the behavior of analog circuits.

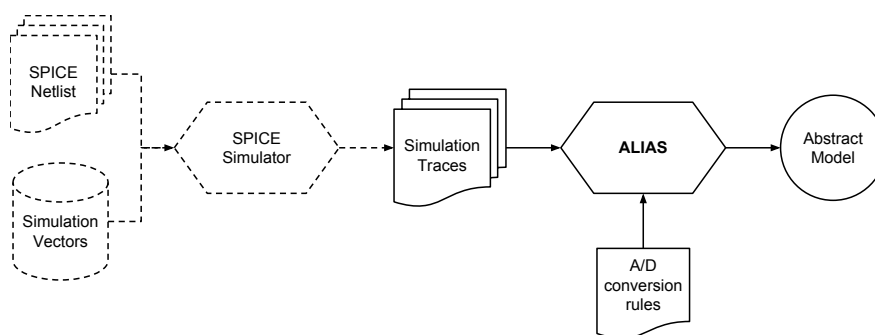


Figure 4.1: An example of a typical ALIAS use flow. Dashed lines indicates optional parts of the flow.

As depicted by Figure 4.1, a typical use-flow of the presented technique would start with ALIAS (analog logical-intent abstraction synthesizer) reading in simulation results (waveforms) and a set of user-specified rules to discretize the waveforms, process them and result on an abstract model in the form of a Verilog or a DOT format file. Optionally, if analog waveforms are not available, then this set of steps would be preceded by SPICE simulations of the analog block to be abstracted against a set of input *stimuli*.

4.2 A walkthrough of ALIAS

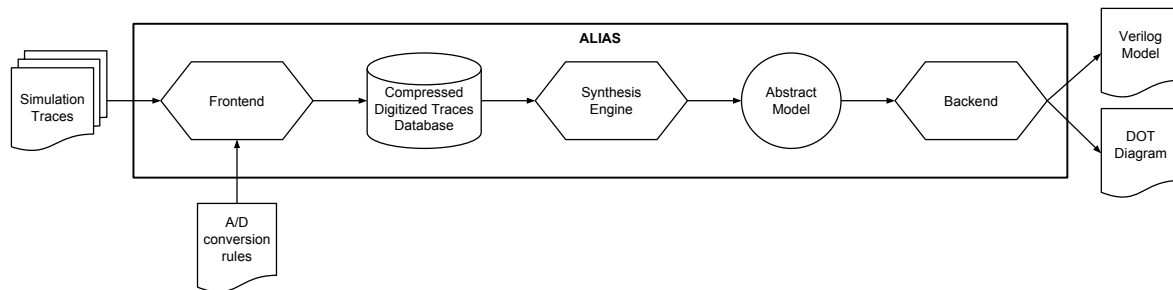


Figure 4.2: System modules data flow.

Architecturally speaking, the system is divided into four main components: **Frontend**, **Discrete Trace Database**, **Synthesis Engine** and **Backend**. The data flow among these components is illustrated by Figure 4.2.

Starting at the Frontend, or Analog Waveform Trace Processor (AWTP), a set of analog waveforms is input to the system along with a specification on how to discretize in space and time analog data.

As digital data is produced by the AWTP, the Discrete Trace Database (DTD), streams this data in and stores it along a description of all nets in the digital samples for current and later executions of ALIAS.

Next, on the Synthesis Engine, the system proceeds on processing digital data coming from the DTD, by detecting repeating patterns, to then create an abstract model of the discretized traces.

Finally, this abstract model proceeds to the Backend, named Model Compiler, which, based on the created abstract model, will produce a Verilog model, to be used on any digital verification environment and DOT models, providing a high-level diagram view of the analog circuit behavior from a digital stand point.

4.3 Frontend: Analog Waveform Trace Processor (AWTP)

The AWTP is in charge of reading analog waveforms and discretizing continuous space/time data based on a parsing specification. This specification informs the parser a time step t , which determines the sampling interval, an offset o , which tells the parser at which point it should start to sample, and a set of rules Φ to convert analog voltage ranges into discrete data. Figure 4.3 exemplifies the parsing process.

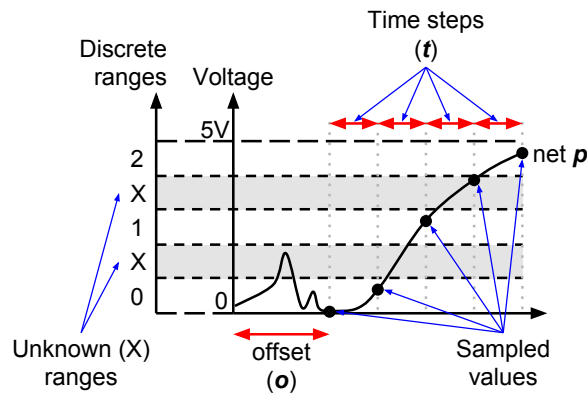


Figure 4.3: An example of space/time discretization of an analog trace, resulting on the sequence 0, 0, 1, X, 2 for some net p .

To define the time-step to be used, we borrow the frequency definition of the digital circuit the targeted analog circuit is going to be verified with.

To sample data, the parser selects the data point nearest to the current time-step. Although simple, this works for all simulations for analog circuits tested to be paired with a digital circuit working at a specific frequency. If that is not the case, per Nyquist-Shannon theorem [Shannon, 1998], the SPICE simulation data points should be at a time-step t_a that is at least half the time-step t_d of the interfacing digital block, that is $t_a \leq t_d/2$. Figure 4.4 illustrates the sampling process on a scenario which $t_d = 4 * t_a$.

Each group G_p of rules defines all digitally known/stable values for a net p given an analog/continuous value, through a mapping $\mathbb{R} \rightarrow \mathbb{Z}$. Therefore, if any analog value input to the parser matches a given range, the corresponding output integer value will be output. Otherwise, a symbol X , representing an unknown state will be reported. Therefore, given an analog value a_p of an arbitrary net p , for $s_i, e_i \in \mathbb{R}$, $z_i \in \mathbb{Z}$ and

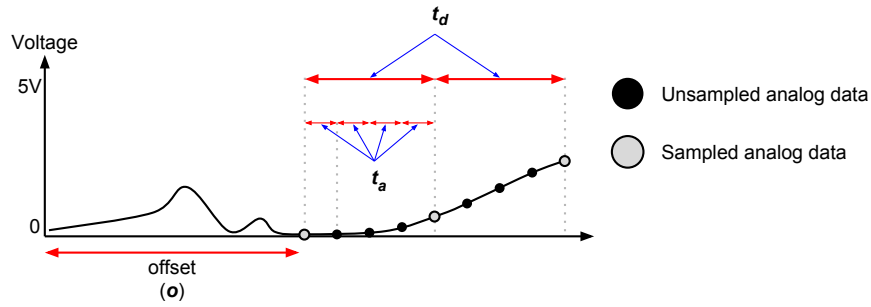


Figure 4.4: An example of sampling analog samples. t_d is the digital time-step and t_a the analog time-step. On the example, $t_d = 4 * t_a$.

$n = |G_p|$, its discretized value d_p is defined by:

$$d_p = \left\{ \begin{array}{l} z_1 \quad \text{if } s_1 \leq a_p \leq e_1 \\ \vdots \\ z_i \quad \text{if } s_i \leq a_p \leq e_i \\ \vdots \\ z_n \quad \text{if } s_n \leq a_p \leq s_n \\ X \quad \text{otherwise} \end{array} \right\} G_p \quad (4.1)$$

This way, Φ can be defined as the set of all groups of rules defined for all nets N under analysis. In short:

$$\Phi = \{G_p \forall p \in N\} \quad (4.2)$$

As data is sampled on steps, each data point can be seen as a snapshot of the circuit state at a given step under a digital perspective. Following this reasoning, a snapshot is, in essence, a tuple with discrete values for all circuit nets under analysis.

Example 4.3.1. On analog/digital interfaces, a common element is the delay line, which is a way of delaying the propagation of a signal. One model of delay line, is a resistor-capacitor (RC) based delay, as depicted by Figure 4.5, which receives a signal *in* and produces a delayed output at *out*.

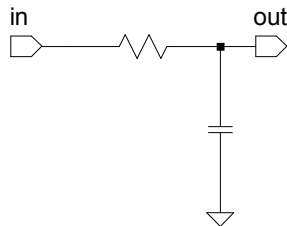


Figure 4.5: RC delay line.

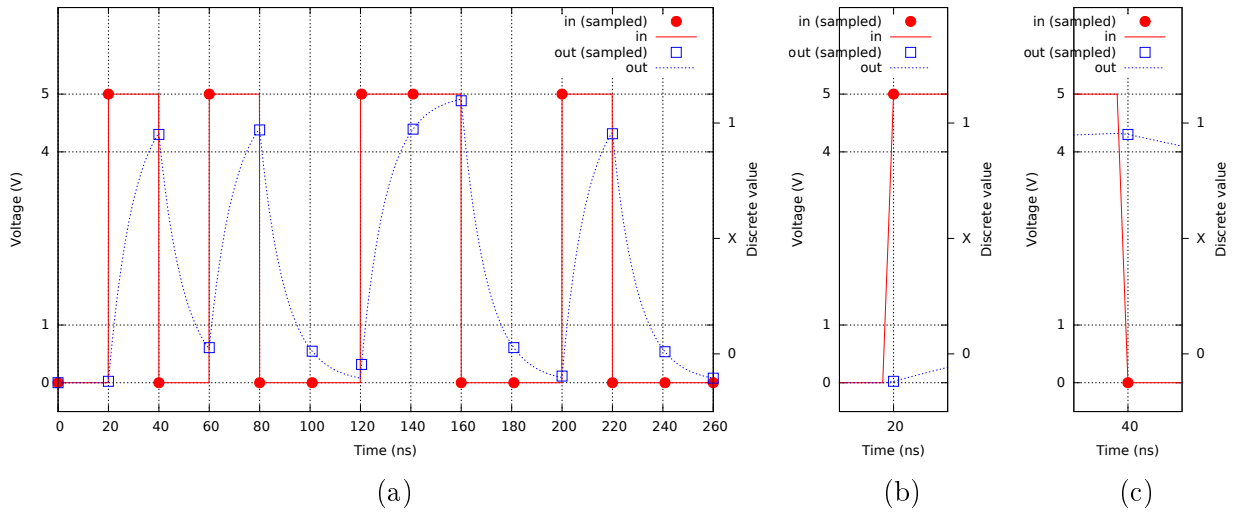


Figure 4.6: RC delay line SPICE simulation trace showing sampled data (a) and detailed sampling sections at 20ns (b) and 40ns (c).

With $\Delta V = 5.0v$ between *in* and ground, the resistance $R = 1.0\Omega$, and the capacitor valued at $C = 0.01\mu F$, within $20ns$ ($2RC$) the capacitor would reach 86% of charge, reaching about $4.32V$, or about 8% above $4.0V$, which we intend to use as threshold for a logic 1. With this setting, for a digital system working at 50MHz, that is, a clock cycle period of $20ns$, *out* would be noticed with a delay of one clock cycle in relation to *in*. Figure 4.6 exercises this delay behavior.

Aligned with the digital system frequency, *in* has a rise/fall time of $0.1ns$, as shown in Figure 4.6b and 4.6c, allowing the time-step t to be defined to exactly $20ns$ and the offset $o = 0$. Given these settings and the rules:

$$\Phi = \{G_{in}, G_{out}\} = \left\{ \left\{ \begin{array}{l} 0 \text{ if } 0.0 \leq in \leq 1.0 \\ 1 \text{ if } 4.0 \leq in \leq 5.0 \end{array} \right\}, \left\{ \begin{array}{l} 0 \text{ if } 0.0 \leq out \leq 1.0 \\ 1 \text{ if } 4.0 \leq out \leq 5.0 \end{array} \right\} \right\} \quad (4.3)$$

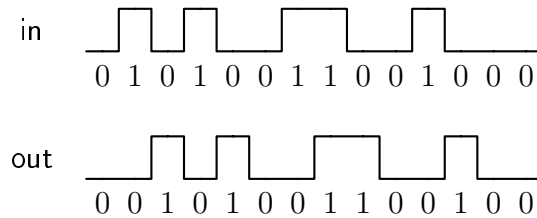
The resulting samples, demonstrating the delay behavior, in the format (d_{in}, d_{out}) , are illustrated in Figure 4.7a, and these same samples as a digital trace in Figure 4.7b.

4.4 Discrete Trace Database (DTD)

The DTD is a drop-in structured storage module, which streams in and handles the persistence of data coming from the Frontend. As state tuples (snapshots) are fed to the DTD, it stores the sequential data as a trace on special entities, called databases. As in traditional database systems *schemas*, the proposed module defines for each collection of traces a *relation*, ie. a formal description of its attributes (a circuit's nets).

[(0, 0), (1, 0), (0, 1), (1, 0), (0, 1), (0, 0), (1, 0), (1, 1), (0, 1), (0, 0), (1, 0), (0, 1), (0, 0), (0, 0)]

(a)



(b)

Figure 4.7: RC delay line samples (a) in the format (d_{in}, d_{out}) and digitized trace (b) from Figure 4.6.

The main reason behind this module is to avoid re-parsing analog simulation traces, which are usually big¹, every time one would try to create or enhance a model with more or less data, thus enabling faster incremental workflows.

In general, analog simulators export traces in text or binary format [Nenzi and Vogt, 2014; Waller, 2010; Tuinenga, 1995], including all information of each simulation point in time, containing the current simulation time as well as the real value of all simulated variables. Although human readable, text-based formats tend to consume more space than binary formats, which, in a general way, are simulator specific and doesn't have any public documentation on how to read them.

On the digital world, one of the most popular formats for exporting waveform data include the Value Change Dump (VCD) [IEEE, 2001] and the Fast Signal Database (FSDB) [Synopsys Inc., 2013] format. On the opposite direction to analog simulators, instead of capturing all data at each simulation step, the VCD format captures only changes made to variables values. To save space, each variable is identified with an arbitrary set of characters, thus, when reporting a change in a point in time, the identifier is used along with its corresponding value.

Even with the technique of reporting just changes over time, the text-based format often results in large files for long and high activity simulation traces. In face of this facts, formats like the binary FSDB gained a lot of popularity, reaching up to 50x [Synopsys Inc., 2013] less space than a VCD file.

Inspired on the VCD format pattern of reporting just changes and on the binary

¹As a reference, at a $1s/1ps$, on the worst case scenario, one can get up to 10^{12} \mathbb{R} data points for a single variable, which means that for a 1s simulation you would have around 7.5GB of raw data.

approach of the FSDB format, we present a binary structured format, called Digitized Trace Database Format (DTDF). Example 4.4.1 illustrates the concept of just reporting changes in a trace.

Example 4.4.1. A traditional way of reporting a simulation step values is:

$$(step, (v_1, d_1), \dots, (v_i, d_i), \dots, (v_n, d_n))$$

With v_i being a variable and d_i its value on the given step. Thus, a valid example of a trace composed by two signals a and b following this step format could be:

$$T = [(1, (a, 0), (b, 0)), (2, (a, 0), (b, 1)), (3, (a, 0), (b, 0)), (4, (a, 0), (b, 0)), (5, (a, 1), (b, 0))]$$

However, if just changes in variables values are to be reported, this same trace can be reduced down to:

$$T = [(1, (a, 0), (b, 0)), (2, (b, 1)), (4, (b, 0)), (5, (a, 1))]$$

To understand the resulting trace, one can observe that between steps 1 and 2 only b changes its value, thus, the second step can be resumed down to $(2, (b, 1))$. Next, as on step 3 no changes are observed on the variables values, that step can simply be removed from the trace. Finally, for steps 4 and 5, the same pattern of step 2 is applied.

A DTDF base is composed by two files, the first storing the base schema, and the second is a binary file, containing the traces stored. The schema file is a text file, and states each variable's name, size (in bits) and direction (input, output or both).

The binary database file, described in Figure 4.8, is informing the number T of traces as well as the number of bytes S_i consumed by each one of the traces. The second part holds all the trace data in blocks, one for each trace.

At each trace block, as shown in Figure 4.8, each non-empty step is represented as a payload containing a header, stating the step/cycle number C_i and how many changes (w) the x th step contains. After this header, given the inherently irregular nature of a change-based trace, to store just observed changes within the x th step/cycle C_x , as shown in Example 4.4.1, it is necessary to identify the i th variable v_i being changed along with its value d_i . Thereafter, a tuple in the format (i, d_i) is stored for each variable value changed. By default, the first step will always contain the values for all variables in the trace.

Space-wise, if all variables stored in the binary database file have the same size, then all tuples (i, d_i) will consume the same space s . Moreover, if the average change

ratio per cycle is r and the set of all variables v_i within the trace is V , the space consumed by each trace within a DTDF file is $E_{\Delta BIN} = O(r * |V| * s)$, highlighting that the consumed space is directly tied to the number of changes at each cycle.

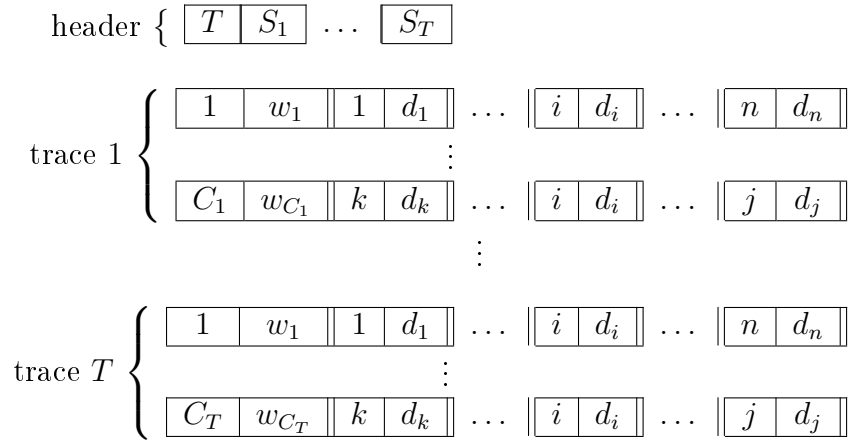


Figure 4.8: DTDF storage organization scheme. On the figure, $1 \leq k \leq i \leq j \leq n$.

4.5 Synthesis Engine

To recall our progress so far, the system's frontend (AWTP) discretized an analog waveform and stored it into a DTD trace set. Thereon, the core component of the system, the Synthesis Engine, will act on.

The general idea of the synthesis engine module is to try to identify a repeating behavior in a trace coming from DTD. Based on Isaksen and Bertacco [2006] work for identifying transactions in digital systems only, we broaden the scope to AMS circuits, as many of such circuits also have a repeating behavior, and use a similar algorithm for detecting such patterns.

More formally, given the language L of a digitized trace, this component aims to identify a language L' such that, as introduced in Definition 2.2.4 on Chapter 2, $L \subseteq L'^*$.

For the sake of simplicity, given the concept of an alphabet, all trace unique state tuples are labeled against an unique symbol, composing an alphabet Σ . Thus, each trace can be seen as a language L composed by one single long string. Example 4.5.1 demonstrates this procedure.

Example 4.5.1. From Figure 4.9 trace, the list of unique state tuples are $U = [(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1), (3, 0)]$. There, the system will create an alphabet Σ with $|U| = |\Sigma|$ composed by arbitrary symbols.

[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)(2, 0), (1, 1), (2, 1), (1, 0), (0, 1), (3, 0)]

Figure 4.9: A simple trace as list of state tuples.

If $\Sigma = \{a, b, c, d, e, f, g\}$, a valid mapping for each tuple in U to a symbol in Σ would be:

$a : (0, 0)$
 $b : (0, 1)$
 $c : (1, 0)$
 $d : (1, 1)$
 $e : (2, 0)$
 $f : (2, 1)$
 $g : (3, 0)$

Thus, resulting labeled trace string will be $abcdefedfcbg$, and, as a result, the trace language $L = \{abcdefedfcbg\}$.

Given L , the goal of the core algorithm of this module is to generate a language L' that generalizes L . In order to do so, L' is built so that each string identifies one repeating behavior in L . More formally, given all strings a circuit can produce in the form of a language L_C , and one simulate the circuit to a set of *stimuli* that yields in $L \subset L_C$, the goal is to have:

1. $L \subseteq L'^*$, or ideally, $L \subseteq \bigcup_{i \in K} L'^i$ with the set of integers K as small as possible;
2. More importantly, L'^* as close as possible to L_C .

In order to generalize, the algorithm tries to recognize partition points, denoting settling states, within the trace string, referred to as *boundaries*. Once these boundaries are defined or recognized, the trace is cut into smaller sub-strings. These are then further reduced by folding repeating sequences with the string operator $(.)^n$. Then, if a substring a is found to be a suffix of a bigger string b , thus converging to the same settling state, b can be further divided, as before a happening in b , b converged first to another settling state. This process is repeated until no new boundaries are found and no further optimizations can be performed over the set of substrings. Example 4.5.2 provides an overall view of the process.

Example 4.5.2. Given a language $L = \{dabbcbcd\}$ from a trace, if d is selected as boundary, we would initially get the set of strings $\{abbcd, bcd\}$. As b repeats twice in $abbcd$, it can be compressed down to ab^2cd . As bcd is a suffix of $a(b)^2cd$, a can be taken as boundary, which will

then produce two new strings from $a(b)^2cd$: a and $(b)^2cd$. By taking strings without its loops annotations, we can detect that $(b)^2cd$ and bcd are equal from that perspective. From that standpoint, we create a new string $(b)^{1,2}cd$, to replace both, thus yielding in $L' = \{a, (b)^{1,2}cd\}$. As one can verify, $L \subset L'^3$.

Besides manual selection, there are several ways to automatically identify the first boundary to be used to partition the trace. Aiming to allow a better automation at larger scales, we select two simple techniques: selecting the symbol that repeats first; and finding the symbol which repeats with the largest interval. The main reason behind the first method is its simplicity and the fact that this is the strategy adopted by [Isaksen and Bertacco, 2006]. As for the second, we aim to capture and create large strings on a first pass. Both techniques are exemplified in Example 4.5.3.

Example 4.5.3. From a string $adadababcbda$, the symbol that repeats first is a . Using a as a cut-point, the resulting strings would be $\{da, ba, bcda\}$. However, by using the symbol that repeats with the largest interval, d would be the resulting partition point, and the resulting substrings $\{ad, ababcd, a\}$.

The algorithm, presented in Algorithm 1, starts by labeling (line 1) all state tuples in the digitized trace to a set of symbols in Σ for each unique state tuple. Next, two sets of boundaries are defined, BS and NBS (lines 2-3), with BS initialized with the first boundary to be used to partition the trace. After that, the trace will be cut using such boundary set elements (line 7). Each resulting string will then be simplified by folding loops within the transaction (lines 8-10). As loops are fold, the algorithm will also merge compatible strings, that is, strings that share the same structure (same loop locations) or have no loop at a region where another string to be merged has. Finally, the algorithm tries to find strings that are contained inside others² and, if found, it will add to the set of new boundaries the last data point before the smaller string matches the bigger one (lines 11-15). The main loop (line 5-16) will repeat until no other distinct transaction is found within another one.

The loop folding algorithm (called from line 9) employs a non-recursive folding approach, using a simple two-step folding strategy. On the first phase, the algorithm tries to find the largest subsequent repeating sub-string s , and folds it in half, thus, from a string with the sequence $\dots xssy\dots$, the resulting folded string would be $\dots x(s)^2y\dots$. After folding, the algorithm never revisits a folded string section again, and will keep

²If a string is found inside another but the point of cut is within a loop, the algorithm will ignore that string unless the given symbol inside the loop is its last. The reason for that is that loops often can mean the number of states until a certain settling state can be reached. Also, by splitting a loop open, we are prone to generating more states and having a large $|K|$ before L can be matched. Thus, we opt to not cut on loops.

Algorithm 1 SynthesizeModel

Require: Digital trace data T_{raw}

```

1:  $T \leftarrow$  label  $T_{raw}$  against  $\Sigma$ 
   #  $BS$  and  $NBS$  stores current and new boundary sets
2:  $BS \leftarrow$  extract initial boundary from  $T$ 
3:  $NBS \leftarrow \emptyset$ 
   #  $S$  is the set of strings
4:  $S \leftarrow T$  string # Start with  $T$ 's single string
5: while  $BS \neq NBS$  do
6:    $NBS \leftarrow BS$ 
7:   partition  $S$  at  $BS$ 
8:   for  $s \in S$  do
9:     fold  $s$  loops
10:  end for
11:  for  $(i, j) \in S, i \neq j$  do
12:    if  $j$  is suffix of  $i$  then
13:       $NBS \leftarrow NBS \cup \{(i - j) \text{ last state tuple}\}$ 
14:    end if
15:  end for
16: end while
17: return  $S$ 

```

on searching for loops on the string remainder portions. This way, after all repetitions are found, the algorithm checks if the folded strings are in fact repetitions of a smaller pattern. Thus, given a folded loop sub-string s , if a pattern p repeats from the first symbol of s until its end, s will be further reduced and the loop number of repetitions multiplied by $\frac{|s|}{|p|}$. The main reason behind this strategy is to avoid spending time on folding intricate loops, and thus, yield on simpler substrings. Example 4.5.4 demonstrates this approach and compares it against a recursive complete approach.

Example 4.5.4. For a string $abbcdabbcdffffffffff$, our proposed approach would start by folding the largest repeating sub-strings, thus yielding in $(abbccd)^2e(ffff)^2$. Then, it would identify that $(ffff)^2$ is composed by a smaller repeating pattern, on this case a single symbol, “ f ”, thus resulting in the compressed loop $(f)^8$. Thereafter, the final compressed string would be $(abbccd)^2e(f)^8$. On the other hand, a recursive full height approach would produce $(a(b)^2(c)^2d)^2e(f)^8$, at the cost of recursively folding already fold strings.

Example 4.5.5. On Example 4.3.1 from an analog delay circuit based on a simple resistor-capacitor circuit, depicted in Figure 4.10, the resulting digitized trace and samples are presented back in Figure 4.11.

Given that the trace has only 4 different state tuples — namely $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$ — line 1 of Algorithm 1 will label with 4 different symbols. For simplicity reasons,

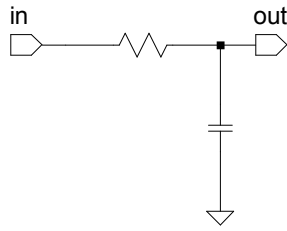
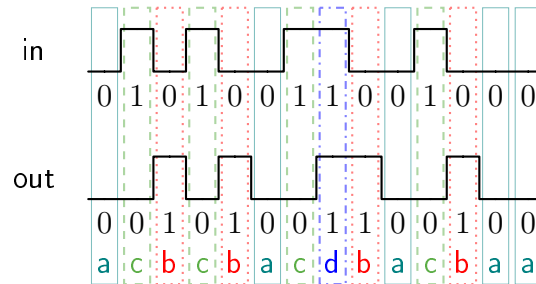


Figure 4.10: RC delay line Example 4.3.1.

[(0, 0), (1, 0), (0, 1), (1, 0), (0, 1), (0, 0), (1, 0), (1, 1), (0, 1), (0, 0), (1, 0), (0, 1), (0, 0), (0, 0)]

(a)



(b)

Figure 4.11: RC delay line samples (a) in the format (d_{in}, d_{out}) and labeled digitized trace (b) from Example 4.3.1.

consider that the selected symbols were a , b , c and d , mapping to:

$$\begin{aligned} a &: (0, 0) \\ b &: (0, 1) \\ c &: (1, 0) \\ d &: (1, 1) \end{aligned}$$

Thus, the resulting alphabet would be $\Sigma = \{a, b, c, d\}$. Using the presented mapping, based on the trace data, the resulting string will be $acbcacdbacbaa$, as depicted in Figure 4.11b. Thus, the trace language is defined as $L = \{acbcacdbacbaa\}$. Using the symbol that repeats with the largest interval (a) as boundary (line 2), would result in the substrings: $\{cbcb, cdba, cba, a\}$ (line 7). Taking all strings individually, the loop folding (line 9) outcome would be $(cb)^2a, cdba, cba, a$. As each string is compressed by the loop folding algorithm (lines 8-10), compatible strings are merged, $(cb)^2a$ and cba will be combined into $(cb)^{1,2}a$. Therefore, we now have the strings $\{(cb)^{1,2}a, cdba, a\}$. When reaching lines 11-15, as a is a viable suffix of $(cb)^{(1,2)}a$ and $cdba$, b will be a new boundary. After partitioning again, the

resulting segments will be $\{(cb)^{1,2}, cdb, a\}$. As no new boundaries are detected, the algorithm finishes with $L' = \{(cb)^{1,2}, cdb, a\}$, and $L \subset L'^3$.

4.6 Backend: Model compiler

Once the synthesis engine has produced a language L' that generalizes a language L , the model compiler will then create a semantic model for L'^* . Based on this model, the compiler will then output it to either a Verilog file, to be used on simulation and formal verification environments, or a DOT diagram/graph, to allow a better understanding of the abstracted system's states and transitions.

4.6.1 Automaton models

As each string in L' is, in essence, a chain of circuit states, to create L'^* we need a way to go through strings in L' in a repeating manner. For that, we present two automaton models, the first named **simple** and, the second, **parallel** automaton model.

4.6.1.1 Simple automata

The first automaton, models an automaton A_s , with each string as clustered a chain of nodes connected to each other. Within a cluster, each node n_i corresponds to a symbol s_i at the position i of a string. Besides nodes coming from strings, one extra node, called *start*, is added, corresponding to the automaton initial state. This node has an edge for each string in L' , and for each string in L' an edge back to the initial state. For each string node, an error edge going back to the node itself, is created. Besides these edges, each node has an edge pointing to the next symbol in its string and, at the last symbol within a loop, an edge back to the loop's start node. Figure 4.12 illustrates this configuration.

As the abstracted circuit is composed by inputs and outputs, each string symbol is composed by an ingoing and an outgoing part. Therefore, a symbol's ingoing portion is matched against the circuit's inputs, the automaton will proceed to that state and emit the outgoing part of the symbol to the circuit's outputs. However, if no next symbol ingoing part is matched, the automaton will remain on the current state until the symbol ingoing part is matched. Moreover, as strings can contain loops that can repeat at different amounts, and as from the start condition it is possible to go to any string initial symbol, A_s is a non-deterministic finite automaton (NFA).

³For the sake of simplicity, loops and error edges were not included.

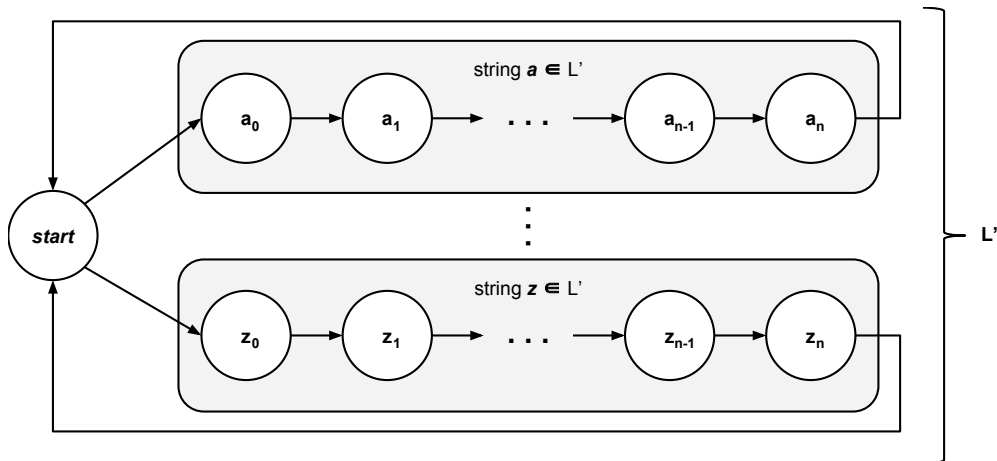


Figure 4.12: Configuration of simple automata³ A_s .

From A_s , we can infer three properties:

1. if an oracle always chooses the correct next state, if an invalid string s is input, $s \notin L'^*$;
2. every path in A_s is a string in L'^* as long as an error edge is never taken;
3. if a set S containing all paths in A_s exists not taking error edges, $S = L'^*$.

Example 4.6.1. Taking back Example 4.5.5, the synthesis algorithm produced $L' = \{a, (cb)^{1,2}, cdb\}$. Thus, the simple automaton A_s for L' is:

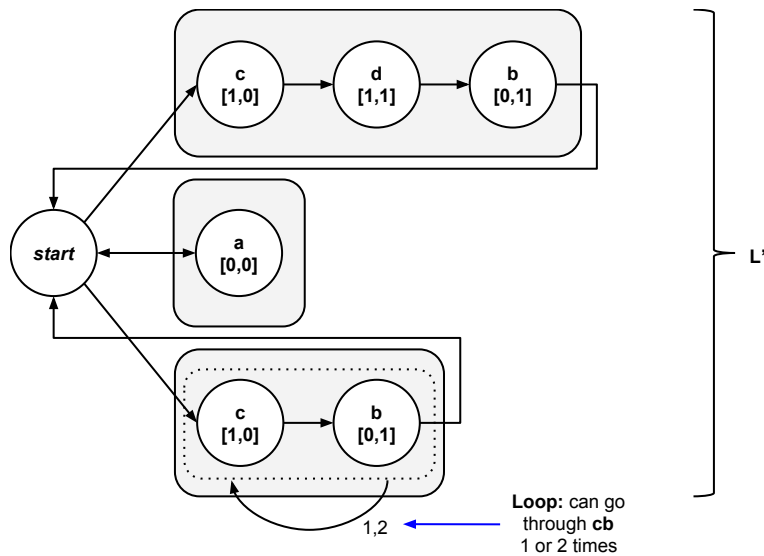


Figure 4.13: Simple automaton model for Example 4.5.5 language L' .

4.6.1.2 Parallel automata

As the simple automaton, a parallel automaton A_p also models each string from L' as a clustered chain of nodes interconnected with edges, with each node n_i mapping to the symbol i th symbol from its corresponding string. The key difference is that each string is seen as an individual automaton, referred to as sub-automaton, executing in parallel and all clusters outputs combined and output by an arbiter (an oracle). Also, the initial state for each sub-automaton is its first symbol, therefore, the last node has an additional edge going back to that state. Identical to the simple model, the last symbol on a loop, has an edge back to the loop's first node. Differently from the last model, each node has an error edge, which return to the node cluster first node. At the cost of being more complex, this organization can be compared with a more compact way of multiplying the number of states of the simple automaton model.

After the simple model, for sub-automaton to transition between states, the ingoing part of a symbol (the node) must match the circuit's inputs. When this ingoing portion is not matched, the error edge is taken, and the given automaton returns to its initial state.

The main component of this model is the arbiter, or oracle, which is responsible for orchestrating outputs coming from the underlying string automaton. The arbiter starts by first selecting a valid string to execute, that is, a node cluster which its initial node symbol matches the circuit input. If no valid string is available, the arbiter waits until one becomes valid. Once a string is selected, the arbiter emits the outputs coming from the selected string until either the current string has been completed (the automaton returned with no errors to the initial state) or an error on that string is detected. If an error is detected, the arbiter will then return to its initial state, selecting another valid string to pick its output from. Figure 4.14 exemplifies this organization.

By enabling the oracle to change its decision after a string has already been selected as the current string, we aim to approximate to a NFA with an oracle that tries to always select the best string to emit output symbols from.

Example 4.6.2. Taking back Example 4.5.5, the synthesis algorithm produced $L' = \{a, (cb)^{1,2}, cdb\}$. Thus, the parallel automaton A_p for L' is:

Similar to A_s , from A_p , we can infer three properties:

1. if the arbiter always chooses the correct string and an oracle always chooses the right next state within a string, if an invalid string s is input, $s \notin L'^*$;

⁴For the sake of simplicity, loops and error edges were not included.

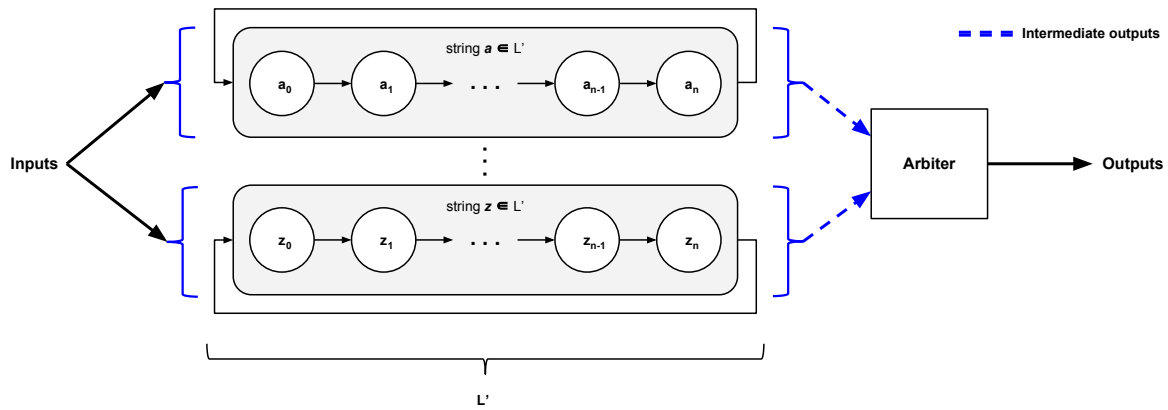


Figure 4.14: Configuration of automaton in parallel⁴ A_p .

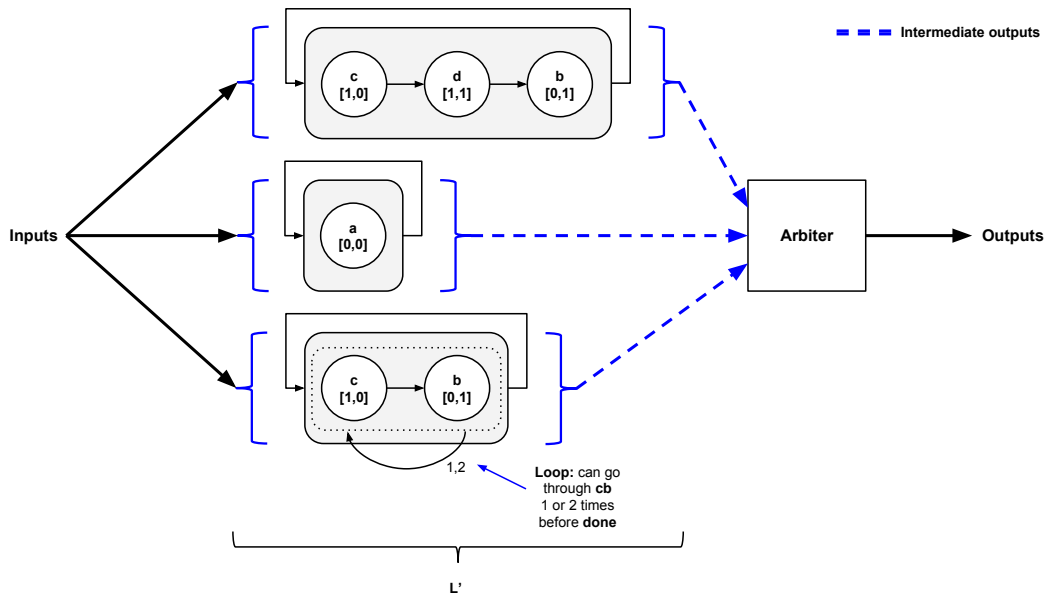


Figure 4.15: Parallel automaton model for Example 4.5.5 language L' .

2. if an error edge is never taken, every path in A_p is a string in L'^* ;
3. if a set S containing all paths in A_p not taking error edges exists, $S = L'^*$.

4.6.1.3 Model compatibility and comparison

Given an oracle that always makes the right choices for A_s and A_p , as previously exposed, both models are compatible with L'^* , thus, are also compatible with the language L the model is based on, and languages like nL , as $nL \in L'^*$.

Considering a standard simulation environment, the automaton cannot know future steps, thus, it can only make its decisions based on either present or past data.

As it happens, a wrong decision can have a considerable impact on the automaton accuracy.

To cope with that, the simple automaton model, in face of an unexpected input, will remain on the current state, while continuing to issue the current symbol outgoing part as output, until a match with the circuit input value happens. With the upside of being simple and not requiring any additional logic or signal to represent the error edge being taken, once an invalid input signal is perceived, it means that the wrong string (a node cluster) was picked for execution, if the input language is compatible with L^* . If so, several other valid input sequences that would match other strings could be passing by the automaton, meaning that the automaton would be stalled until a sequence with a match on the current symbol reaches the automaton.

The parallel model, however, given an input that is not recognized by any of its sub-automaton, will, as on the simple model, repeat the current node symbol outgoing part as output. As in this case all sub-automaton will go back to their initial states, the arbiter will exit this generalized error state once any valid incoming substring starts to be recognized by any of its sub-automaton. On the upside of this approach, even if the wrong node cluster is selected for execution, as all sub-automaton execute in parallel, a not failing one will be picked as soon as the current fails. Thus, when compared to the previous approach, this model has more chances of resuming a non-error operation faster.

Aside from aiding the automaton to recover from an error state, these approaches aim to allow the automaton to accommodate small variations on the input language, which can exercise behaviors not covered on the language the automaton was built upon.

One important turning point between the proposed models in simulation environments is the amount of non-determinism. On the simple model, when a node cluster has finished its execution, the automaton has to choose between all strings a single string for it to start traversing. As the size of the model grows, this non-determinism can have a deep impact on the model accuracy. On the other hand, the parallel model does not face this same problem, as all strings execute in parallel and a bad choice has a greater probability of being rapidly rectified.

The disadvantage of the parallel approach when compared to the simple one is its bigger footprint, as it will need the number of strings in L' times the number of state tracking control bits in the simple model. More, as on formal verification environments is possible to enforce the model to never take an error edge on the model, the fewer the bits on the model, the faster formal engines will be able to handle the model.

As it is not the intent to generate an output formed by invalid substrings, using

the parallel model on formal engines would inevitably generate an invalid output. This happens as the oracle logic would allow the engine to change between substrings between the execution of a single substring. With a lower amount of non-determinism of the simple automaton, a formal engine would be able to form a valid sequence formed only by valid substrings. Another important factor when dealing with formal engines is that the parallel model requires a larger amount of bits to track the state of each automaton. Given that formal engines targets is usually modeled in a bitwise fashion, a larger amount of bits can mean the non-convergence of a formal task.

With the presented information it is clear that the simple model is the best choice for formal verification environments, whereas the parallel model is better suited for simulation environments.

4.6.2 DOT diagrams compilation

On both cases as the automaton A is a (directed) graph, generating a DOT model is a straightforward direct conversion task, as all the DOT language requires is the declaration of the graph nodes and their connections. As our goal here is to just present the strings node clusters and non-error edges for both automata models, the DOT representation can be generated by a single algorithm.

Algorithm 2, the algorithm iterates over the node clusters (strings) in A (lines 3-19) and first declare that a cluster is being created (line 4). Right after, all nodes in the current cluster c that are not part of a loop are declared (lines 5-9). Next, for each loop, the algorithm will create a subcluster along with the statement of its composing nodes and how many times it can be executed (lines 10-17). Then, the algorithm will close the current cluster (line 18) and proceed to the next cluster. After iterating over all clusters, all edges will declared (lines 20-22) and the algorithm terminates.

Example 4.6.3. For the automaton A_s and A_p from Example 4.6.1 and Example 4.6.2, respectively, Algorithm 2 would generate Figure 4.16 graphs.

4.6.3 Verilog models compilation

Encoding an NFA as a digital system hardware implies that the automaton will be clocked by a clocking signal and will also have a reset condition, into which all relevant registers will be set to an initial state.

To compile both automaton models into Verilog code, the compiler will first start by defining the automaton module interface and signals. Then, it proceeds on encoding the core logic of the automaton.

Algorithm 2 GenerateDOTDiagram

Require: Automaton A # O stores the DOT file being generated

```

1:  $O \leftarrow \emptyset$ 
2:  $O+$  = declare start state
3: for cluster  $c \in A$  do
4:    $O+$  = declare cluster start
5:   for node  $n \in c$  do
6:     if  $n$  is not at loop then
7:        $O+$  = declare node  $n$ 
8:     end if
9:   end for
10:  for loop  $l \in c$  do
11:     $O+$  = declare cluster start
12:     $O+$  = declare loop repetitions
13:    for node  $n \in l$  do
14:       $O+$  = declare node  $n$ 
15:    end for
16:     $O+$  = declare cluster end
17:  end for
18:   $O+$  = declare cluster end
19: end for
20: for edge  $e \in A$  do
21:   $O+$  = declare  $e$ 
22: end for
23: return  $O$ 

```

4.6.3.1 Simple automata

The simple automaton model has three main registers: s_{id} , n_{id} , $start$, $error$. The first register, s_{id} , tracks the current string (cluster) being traversed in the automaton, whereas the second, n_{id} , tracks into which state inside the current string the automaton is. The last two registers tracks if the automaton is either on the *start* state or on the error *state*, respectively. A signal, named *state*, keeps the current symbol value based on n_{id} .

When the circuit is reset, $start$ will be set to 1 and $error$ to 0. After reset, as the clock ticks, the first check the automaton will do is to see if the current string is *start*, if so, it will assign s_{id} to a random string and set $start$ to 0.

On the same clock event, independently from s_{id} value, the automaton will check if the symbol pointed by n_{id} matches the circuit's input signals. If so, the automaton will update n_{id} , reflecting the changes on the signal *state* accordingly. Otherwise, the automaton will set $error$ to 1.

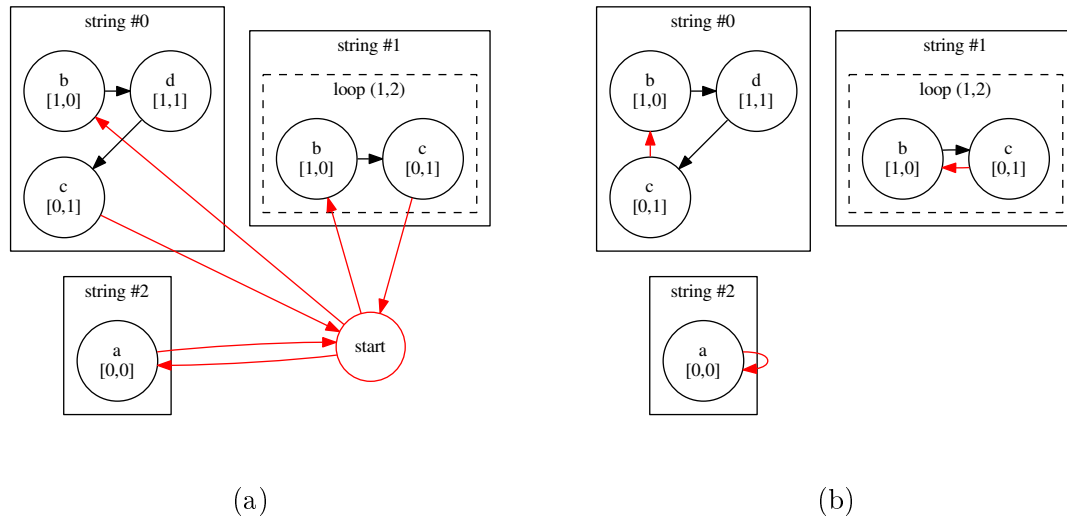


Figure 4.16: DOT diagrams for (a) Example 4.6.1 and (b) Example 4.6.2 automaton. Different edges and nodes between representations in red.

To update n_{id} within a string, when not in a loop, the current state will simply pick the next state. To deal with loops, two extra registers are defined, namely c and c_{max} . The first tracks how many times a loop has been executed, while the second stores the maximum number of times the loop will be executed. To cope with these two registers, when s_{id} is *start*, the automaton will set $c = 0$ and randomly assign a value to c_{max} based on the possible repetition amounts of the first loop in the string.

If when on a loop's the last state, $c = c_{max}$, n_{id} will point to the next state in the current string, c will be set to 0 and c_{max} randomly assigned to the repetition amounts of the next loop in the current string. However, if $c < c_{max}$, n_{id} will be set to the state matching the loop first state and c incremented by 1.

4.6.3.2 Parallel automata

As in the simple model, all sub-automaton has registers to track its state. These registers are n_{id}^i , $error^i$ and $done^i$. The first register n_{id}^i holds which state inside the current string the automaton is. The following register, named $state^i$, keeps the current symbol value. The remaining registers, $done^i$ and $error^i$, tracks, respectively, if the sub-automaton has completed its execution or failed to recognize the current circuit input being passed to the automaton.

After the simple model, all sub-automaton employ the same loop handling mechanism, having the registers c^i and c_{max}^i tracking the loop count and the maximum repetition amount, respectively.

Algorithm 3 GenerateSimpleVerilogModel

Require: Automaton A_s

O stores the Verilog file being generated

- 1: $O \leftarrow \emptyset$
- 2: $O+$ = declare module interface
- 3: $O+$ = declare signals # signals = registers and wires
- 4: $O+$ = map output signals to *state*
- 5: $O+$ = declare clocked block **start**
- # Circuit reset block
- 6: $O+$ = declare reset block **start**
- 7: $O+$ = declare $s_{id} \leftarrow$ **start**
- 8: $O+$ = declare $n_{id} \leftarrow 0$
- 9: $O+$ = declare reset block **end**
- # Circuit non-reset block
- 10: $O+$ = declare non-reset block **start**
- 11: $O+$ = create **start** state block
- 12: **for** cluster $c \in A_s$ **do**
- 13: $O+$ = create c block
- 14: **end for**
- 15: $O+$ = declare non-reset block **end**
- 16: $O+$ = declare clocked block **end**
- 17: **return** O

For the arbiter, two main registers are created. The first, named s_{id} , tracks which sub-automaton output is being emitted as the abstraction output. The second, *last* holds the last state outgoing symbol part. Aside from registers, most of the arbiter logic is combinational. Thus, several auxiliary combinational signals, were used. The first, sc_{id} , tells the arbiter the first sub-automaton that is not failing if the one pointed by s_{id} is. This allow the arbiter to rapidly switch from an error state on the current sub-automaton to the first non failing one. Another key signal is *state*, which tracks the overall automaton current symbol based on s_{id} and n_{id}^i information.

When the circuit is reset, all sub-automaton will set its current state as the string first symbol. Also, a random string will be assigned to s_{id} . After reset, as the clock ticks, the first check the arbiter will do is to see if the current string, that is, the one being pointed by s_{id} has not reached an error condition ($error^i = 0$). If the current sub-automaton is not failing, then the arbiter will continue emitting the output coming from that sub-automaton. However, if $error^i = 1$, the arbiter will automatically pick the first sub-automaton that is not emitting an error, and, with the signal sc_{id} update s_{id} with a non-failing sub-automaton identifier. If all sub-automaton are failing, than the arbiter will continue with the same s_{id} , and emit *last* as output until a sub-automaton

Algorithm 4 GenerateParallelVerilogModel

Require: Automaton A_p

O stores the Verilog file being generated

- 1: $O \leftarrow \emptyset$
- 2: $O+$ = declare module interface
- 3: $O+$ = declare signals # signals = registers and wires
- 4: $O+$ = map output signals to *state*
- 5: $O+$ = declare clocked block *start*
- # Circuit reset block
- 6: $O+$ = declare reset block *start*
- 7: $O+$ = declare $s_{id} \leftarrow$ random sub-automaton id $\in A_p$
- 8: **for** cluster $c \in A_p$ **do**
- 9: $O+$ = declare $n^c_{id} \leftarrow 0$
- 10: **end for**
- 11: $O+$ = declare reset block *end*
- # Circuit non-reset block
- 12: $O+$ = declare non-reset block *start*
- 13: $O+$ = create *start* state block
- 14: **for** cluster $c \in A_p$ **do**
- 15: $O+$ = create c sub-automata
- 16: **end for**
- 17: $O+$ = declare non-reset block *end*
- 18: $O+$ = declare clocked block *end*
- 19: **return** O

starts to recognize incoming *stimuli* again.

For any given sub-automaton to evaluate $error^i$, the arbiter will check if the symbol pointed by n^i_{id} matches the circuit's input signals. If so, the automaton will update n^i_{id} with the next symbol's position. Otherwise, the automaton will set $error^i$ to 1 and set n^i_{id} to 0.

Chapter 5

Case studies

As previously exposed, following the increasing IC complexity, time spent on verification during the execution of an IC project is overcoming development time. Thereon, verification engineers are in constant need of reliable methodologies to bring the circuit being verified into higher (and faster) abstraction levels.

Thus, to verify our proposed solution, we compared the generated simplifications, in terms of quality and speed when compared to a setup using an analog SPICE simulator. We tested our solution against common analog mixed-signal (AMS) circuits. Those being a simple resistor-capacitor (RC) delay line, an integrator and a phase-locked loop (PLL). For every inspected instance, we demonstrate that even being abstract, the generated models were reasonably accurate and far faster than running an analog simulation.

To compare performance, we run an analog simulation over the real analog interfacing circuit; and a purely digital simulation using an ALIAS abstraction instead of the analog circuit.

At each case study, first, a base simulation, producing a language L is performed, from which ALIAS abstractions (simple and parallel models) will be created.

On elaborating the base simulation, we define a metric of *scenarios*, which measures how many different situations the targeted analog circuit has been exercised between rest states. This metric has a direct relation with how many strings the automata will contain.

Given the input stimuli from each case study test benches, we measure performance by simulating the digital abstractions and analog circuit against this sequence of stimuli. On this same line of thought, to measure the quality of abstractions generated by ALIAS, we compare the generated abstraction outputs against the expected outputs for each case. Given this methodology of measuring quality, we compare the

results for each kind of boundary technique selection adopted.

The tests were conducted on a Intel[®] Core[™] i7-2600 @3.4GHz with 8GB of main memory. To perform the tests, two third-party softwares were used, namely the proprietary simulator LTSpiceIV[®] 4.22r [Linear Technology, 2015] was used for analog simulations and the open-source simulator Icarus[®] Verilog 0.9.7 [Williams, 2015], for digital simulations.

5.1 RC delay line

As presented on previous sections, in Example 4.3.1 and Example 4.5.5, a RC delay line, as show in Figure 5.1, is an analog element designed to delay the propagation of the observed value of a given signal.

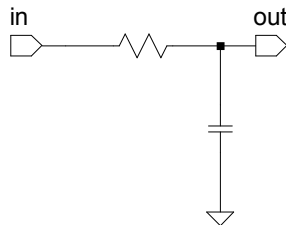


Figure 5.1: RC delay line.

The first set of tests is created after a base simulation. For that, although not covering the entire state space of the given circuit, the analog simulation from Example 4.3.1 was selected, as it covers key characteristics of the delay element operation behavior. Then, using the basic test bench described in the previous section, for simulations of size $100L$ (1400 steps), we analyze the impact on quality for the proposed boundary selection techniques as well as the symbol which we considered to be the circuit resting (idle) state.

As one can check from the resulting string¹ for the selected trace, as shown in Example 4.5.5, using the first repeated symbol as boundary, which did not match the actual circuit resting state (both input and output at zero), the simple model achieved a quality ratio of just 72.79%, while the parallel model got 96.43%. The symbol that repeats with the largest interval, however, which matches the circuit resting state, reached 82.93% of quality in the simple automaton and 100% in the parallel automaton.

As for the automata performance, based the first test bench still, as shown on Table 5.1 and Figure 5.2, the digital simulation using ALIAS abstractions clearly out-paced analog simulations in terms of speed. To further highlight this, at simulations of

¹As shown in Example 4.5.5, a possible string for the example's digitized trace is *acbcbaacdbacbaa*.

size 10^6L (14 million steps), the speedup using ALIAS abstractions (both simple and parallel), is on order of magnitude of 10^3 faster than analog simulations.

As the abstraction quality is key, the simple model reached a quality of nearly² 80%, the parallel model reached 100% of quality, at all tests, as presented on Table 5.2. As the same sequence of strings, and stimuli, is repeated over and over, due to the simple model non-determinism on the transitions between the start node and the first node of each node cluster, it doesn't always select the right next string to execute, thus erring out until the incoming input signal is recognized. Due the small output state space (just two symbols, 0 and 1), repeating the current state until the automaton leaves the error condition is a reasonable approach. The parallel model, however, never has all its sub-automata failing at the same moment, as each part of the input stimuli is part of L' and each element in L' has its own sub-automaton executing in parallel.

Furthermore, if high quality is not of interest, the simple model is the best abstraction candidate, as it is twice as fast the parallel model on bigger cases. However, if quality is key, even with the additional overhead, the parallel model still have a linear complexity, thus scaling to far bigger circuits and simulation traces, which analog simulations wouldn't.

Test size (steps)	Time (s)		
	SPICE	Digital (simple)	Digital (parallel)
14	0.004	0.004	0.004
140	0.006	0.006	0.008
1400	0.037	0.011	0.021
14000	0.439	0.065	0.126
140000	10.539	0.603	1.179
1400000	1117.513	6.101	12.157
14000000	111552.266	58.826	121.800

Table 5.1: RC repeating L tests speed comparison between an analog simulation and digital only simulation using ALIAS abstractions (simple and parallel).

Instead of repeating the stimuli that generated L , the second test flavor is based on the generation of random stimuli, from which traces still with sizes based on L are simulated. Aiming on a greater variety of traces, for each targeted trace size, 5 sets of random input stimuli were generated, which were used as the circuit input in both analog and digital simulations. Finally, the average from the results of each targeted trace size along a confidence interval of 95% are presented.

²Geometric average of quality ratios for the simple model abstraction.

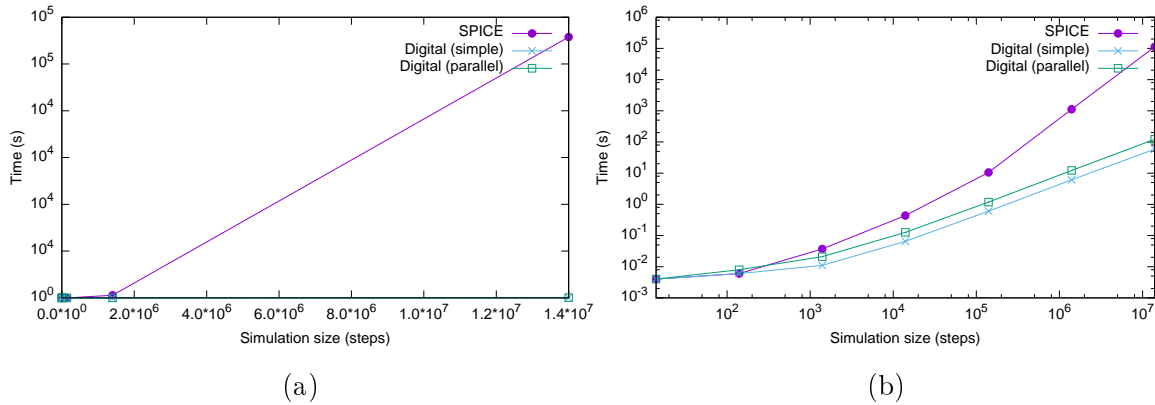


Figure 5.2: RC repeating L tests speed (a) plain and (b) logarithmic scale charts, demonstrating analog simulations exponential trend and the linear behavior of digital only simulations using ALIAS abstractions (simple and parallel).

Test size (steps)	Quality	
	Digital (simple)	Digital (parallel)
14	71.43%	100.00%
140	82.86%	100.00%
1400	83.71%	100.00%
14000	83.14%	100.00%
140000	81.96%	100.00%
1400000	81.06%	100.00%
14000000	81.04%	100.00%

Table 5.2: Quality ratios for RC repeating L tests over ALIAS abstractions (simple and parallel).

As presented in Table 5.3 and Figure 5.3, the random stimuli tests demonstrated a very similar performance when compared to the repeating L tests. As different patterns, like longer repetitions of 0's and 1's, were exercised, in terms of quality, the results were slightly different.

From Table 5.4, the simple automata performed better when compared to the previous set of tests, averaging (geometric mean) nearly 87% in quality. As the simple automaton randomly selects strings (node clusters) for it to execute, on the first set of tests, the chances of always choosing the same strings in the same order, as in the input pattern, decreases considerably as the number of automaton strings grows. Thus, random stimuli experimentally had a better match with the inherently non-deterministic automaton behavior on selecting strings to traverse. The parallel automaton, on the other hand, had its quality decreased, averaging (geometric mean) 99.70% of quality. The main reason behind this fact is that the input pattern is no longer based on L' ,

thus, on some occasions none of the automata will recognize the incoming pattern, which, as a reference, happened at an average of $0.81\% \pm 0.06\%$ of the simulation time with a confidence interval of 95% for a $1000L$ simulation.

Test size (steps)	Time (s)		
	SPICE	Digital (simple)	Digital (parallel)
14	0.004 \pm 0.002	0.004 \pm 0.001	0.005 \pm 0
140	0.007 \pm 0.001	0.006 \pm 0	0.008 \pm 0.001
1400	0.037 \pm 0.002	0.014 \pm 0.002	0.024 \pm 0.003
14000	0.347 \pm 0.003	0.086 \pm 0.001	0.150 \pm 0.002
140000	8.501 \pm 0.02	0.819 \pm 0.007	1.443 \pm 0.011
1400000	956.451 \pm 0.847	8.098 \pm 0.046	14.837 \pm 0.064

Table 5.3: RC random tests speed comparison between an analog simulation and digital only simulation using ALIAS abstractions (simple and parallel).

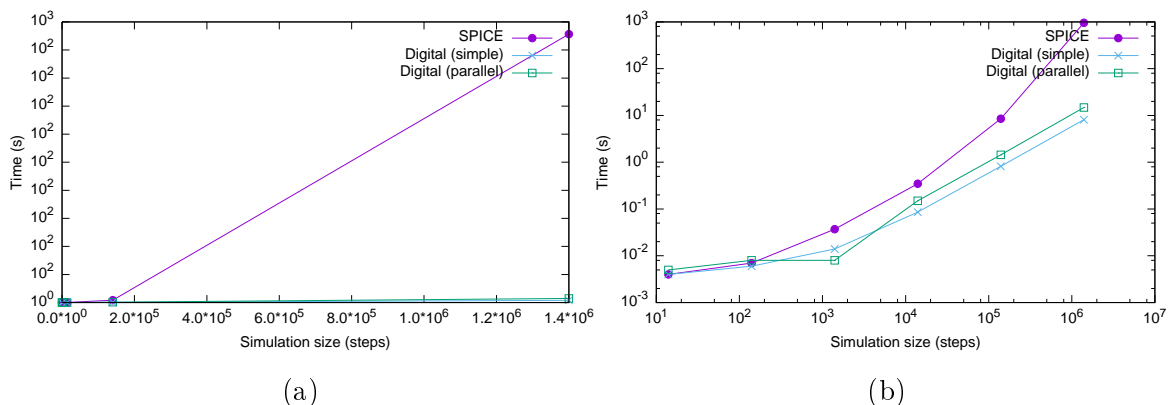


Figure 5.3: RC random tests speed (a) plain and (b) logarithmic scale charts, demonstrating analog simulations exponential trend and the linear behavior of digital only simulations using ALIAS abstractions (simple and parallel).

5.2 Integrator

On this case study we analyze an analog integrator, a component widely used on analog-to-digital/digital-to-analog (AD/DA) converters. The integrator is a circuit that performs the mathematical operation of integration through time with respect to an input voltage. For this case study we analyze an operational amplifier³ (op-amp) based integrator, taking an input with a polarity p , and outputting the integrated signal with a polarity $-p$.

³The more interested reader should refer to Alexander and Sadiku [2008].

Test size (steps)	Quality	
	Digital (simple)	Digital (parallel)
14	91.43% \pm 5.24%	100.00% \pm 0%
140	87.86% \pm 3.76%	99.71% \pm 0.56%
1400	85.74% \pm 1.43%	99.67% \pm 0.56%
14000	86.36% \pm 0.52%	99.58% \pm 0.03%
140000	85.87% \pm 0.12%	99.60% \pm 0.01%
1400000	85.73% \pm 0.05%	99.60% \pm 0.01%

Table 5.4: Quality ratios for RC random tests over ALIAS abstractions (simple and parallel).

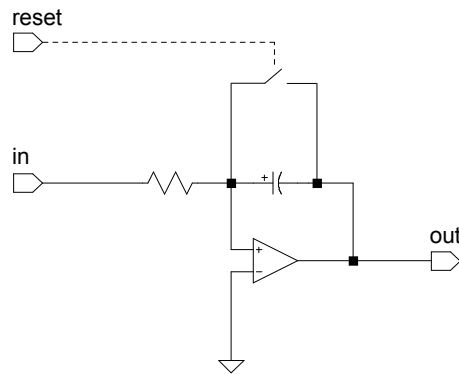


Figure 5.4: Analog integrator circuit.

As shown in Figure 5.4, the analyzed circuit has three main interfacing signals. The first is the signal being integrated, named *in*, the second, *out*, the current integration result, and the last, *reset*, resets the integration result to 0V. Figure 5.5 illustrates a typical operation of an integrator with *in* at a constant $-1V$, depicting the circuit output summing up to 10V, when *reset* acts and sets *out* back to 0V.

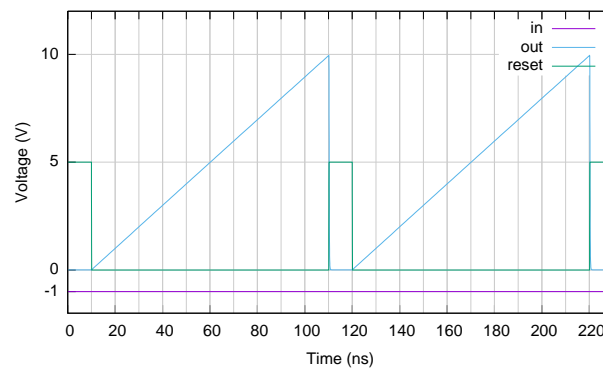


Figure 5.5: Integrator SPICE simple simulation trace.

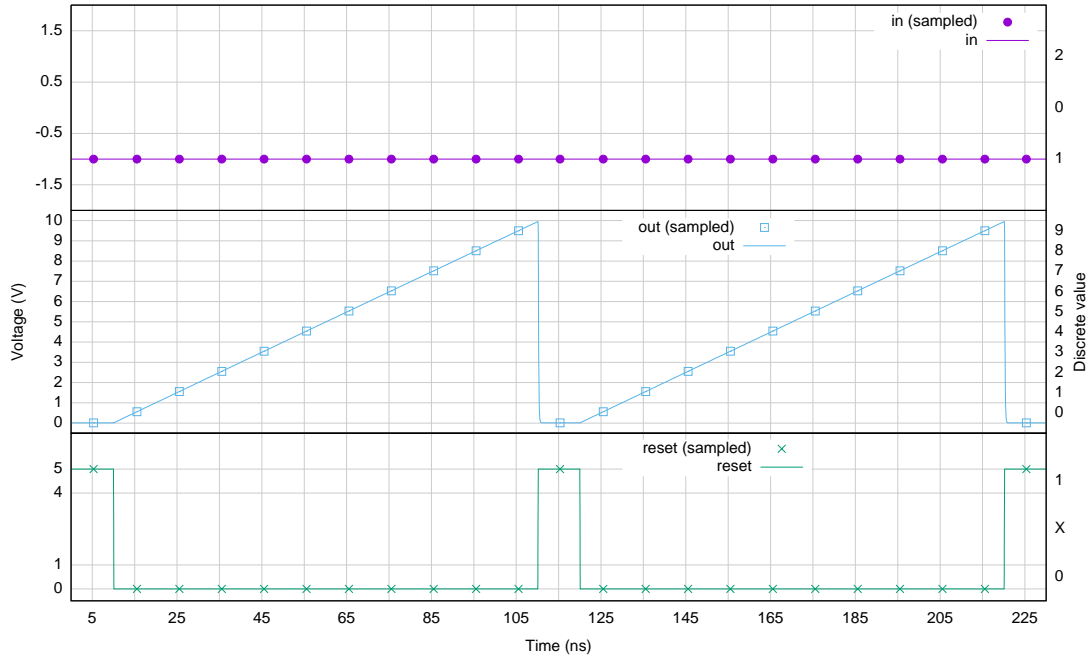


Figure 5.6: Integrator SPICE simple simulation trace showing sampled data.

With this case study we aim to show how important it is to have a good base simulation. We start by creating our first abstractions using the simulation presented on Figure 5.5, exercising just one scenario of behavior (integrating up to $10V$ and resetting at the end). As the targeted analog circuit is sampled at each $10ns$ by a digital circuit, we sample the analog trace at each $10ns$. Figure 5.6 depicts the extracted samples as well as the rules Φ used to convert analog data into discrete values.

As both techniques of boundary selection will end up with the same tuple of values, $(d_{in}, d_{out}, d_{reset}) = (1, 0, 1)$, and this tuple is indeed the system’s rest (idle) state, we proceed on analyzing the quality of the generated abstractions. For the sake of brevity, we attain to this boundary for the remainder of this section.

Starting with the base test bench — created by repeating the input stimuli from which the abstractions were created — from Table 5.5 and Figure 5.7, we observe the same performance trend as in the RC tests, with ALIAS abstractions scaling linearly while SPICE simulations blow exponentially in time complexity.

As the base simulation used to create the abstractions just covered one scenario between rest states, the generated automata had just one string. As there are no loops within the given trace, and as expected, both automata achieved 100% of quality on all sizes. However, when exposed to larger simulations ($1000L$) that applies random stimuli over the signal *reset*, exercising uncovered scenarios, as expected, the quality for this specific case was poor, reaching just 10.49% using the simple automaton and

Test size (steps)	Time (s)		
	SPICE	Digital (simple)	Digital (parallel)
11	0.003	0.005	0.005
110	0.008	0.005	0.005
1100	0.069	0.015	0.014
11000	0.657	0.054	0.063
110000	7.577	0.446	0.574
1100000	273.977	4.461	6.111
11000000	22277.405	44.602	60.746

Table 5.5: Integrator repeating L tests speed comparison between an analog simulation and digital only simulation using ALIAS abstractions (simple and parallel).

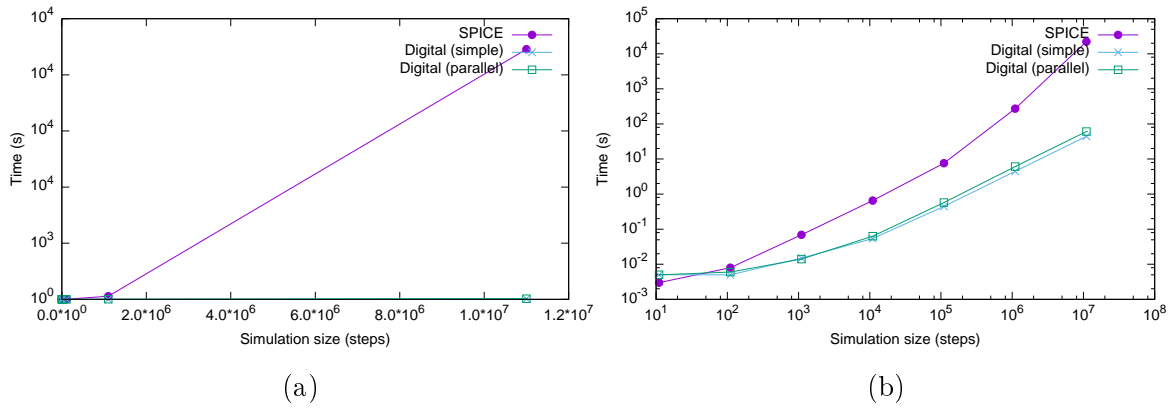


Figure 5.7: Integrator repeating L tests speed (a) plain and (b) logarithmic scale charts, demonstrating analog simulations exponential trend and the linear behavior of digital only simulations using ALIAS abstractions (simple and parallel).

64.60% using the parallel automaton.

Given random simulations constructed to exercise integration scenarios from an interval of 0 to 1V up to intervals of 0 to 11V over the signal *out*, we define 14 scenarios. To create such scenarios, in the opposite direction of the simulation example presented in Figure 5.6 we exercise ranges of voltages over *out* by enabling and disabling the *reset* signal. By doing so, we aim to cover voltage integration operations ranging from 0 to 1V up to intervals from 0V to 11V, supplying in fact more scenarios than the ones present in the targeted random simulations. As shown in Table 5.6 and Figure 5.8, by increasing the number of scenarios covered in the base simulation, both abstractions shows a gain in quality when compared to the simulation with just one scenario. In the simple automaton, the quality reaches up to 25% with 10 scenarios in place. However, from the eleventh scenario on, the increased non-determinism causes its quality to fall to 19.91% at 14 simulation scenarios used. On the other hand, the parallel automaton,

demonstrates an increasing quality as each scenario is added, reaching up to 88.2% with 11 scenarios and stabilizing at that number. As we only are looking at a single input value (the *reset* signal) as a condition to transition in the automaton, on some occasions smaller sub-automata that just erred out or completed its execution may start to recognize a sequence belonging to another longer sub-automata first, and get selected by the arbiter, as the arbiter always selects sub-automata in the same order. As it happens, this is the cause of why a greater level of quality was not achieved on these tests for the parallel automaton.

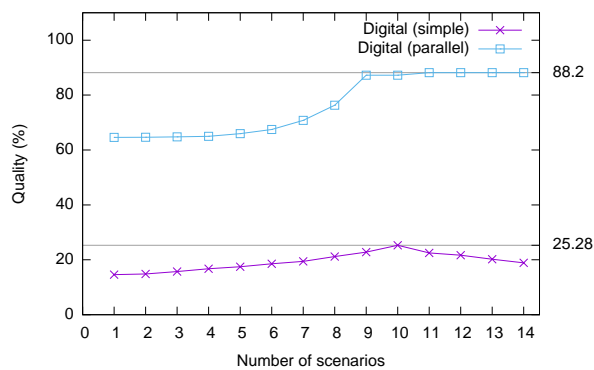


Figure 5.8: Integrator random tests comparing ALIAS abstractions (simple and parallel) quality with the amount of scenarios used to create the abstractions for simulations of $1000L$ sampling points.

5.3 Analog comparator

One of the most popular uses of an analog comparator is to simply compare if the voltage from a source a is lower than another b . Thereon, based on the Linear Technology LT1011 analog comparator [Linear Technology, 1991] the circuit on Figure 5.9 was used. For this circuit, we created an abstraction using a base simulation covering all cases from 0 to 5V with 5 levels, rendering 25 scenarios.

Performance-wise, for a 100000 steps random simulation, the analog SPICE simulation took 696.619s to execute, while the digital simulation using ALIAS abstractions took 0.807s, for the simple automata, and 3.544s, for the parallel model, a speedup of $O(10^2)$ in both cases. Moreover, the quality of the simple automata-based abstraction reached 84.78% while the parallel one, 100%. The high quality achieved on the simple automata (which has 25 possible transitions from the initial state) can be explained due to the fact that even without any reasoning over the states, one has 50% of answering

Number of scenarios	Quality	
	Digital (simple)	Digital (parallel)
1	14.60%	64.60%
2	14.84%	64.65%
3	15.74%	64.80%
4	16.75%	65.00%
5	17.46%	65.98%
6	18.57%	67.47%
7	19.45%	70.79%
8	21.19%	76.32%
9	22.80%	87.24%
10	25.28%	87.27%
11	22.51%	88.20%
12	21.70%	88.20%
13	20.19%	88.20%
14	18.91%	88.20%

Table 5.6: Integrator random tests comparing ALIAS abstractions (simple and parallel) quality with the amount of scenarios used to create the abstractions for simulations of $1000L$ sampling points.

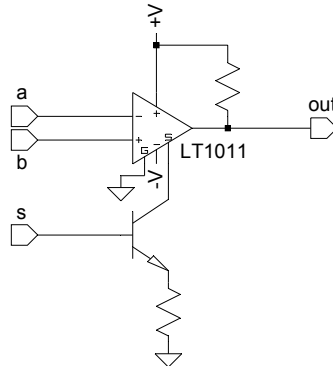


Figure 5.9: Typical analog comparator topology.

the question whether $a < b$, and, as confirmed in simulations, in many situations, this was the case. However, the parallel automata got an exact match on all comparisons.

5.4 PLL

Phase-locked-loops (PLLs) is one of the most essential components on modern ICs and is used on a variety of timing related functions, such as demodulation, clock recovery and clock de-skewing. A PLL, as illustrated in Figure 5.10, is a system that generates

an output signal based on the phase of an input signal. For that, the phase of the output signal is continuously adjusted based on the phase difference between the input and the generated output signal. The main goal of this feedback system is to lock the output phase to a specific relation with regard to the input phase. On synchronization systems, this point can be seen as when both input and output phases are locked in.

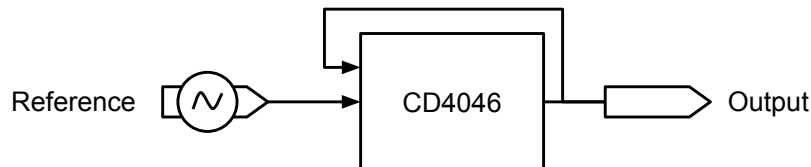


Figure 5.10: Typical PLL-based synchronizer topology.

The PLL circuit selected for analysis is based on the CD4046 [Fairchild Semiconductor, 2003] data-sheet and implements a simple clock synchronizer. For this case study, we take a PLL with a reference clock of $0.2MHz$ as the system input, that is, with this circuit we aim to regenerate a clock signal based on the system input over the output signal. Based on the used PLL specification, we created 18 scenarios for the base simulation, covering an out-of-sync output clock being locked in with the reference one. More importantly, the synchronization behavior was captured using just two of the PLL signals, the reference clock and the circuit output (which was taken as both input and output of the created abstraction). As a basic set of tests, we compare running the analog simulation against an already synchronized output clock and a digital simulation over ALIAS abstractions. Next, we analyze the abstractions behavior over an out-of-sync output clock simulation. As with this test we only aim to verify the ability of the system to synchronize over an external signal, we measure the simulation in terms of the reference clock periods. The results, for 1000 periods simulations are presented in Table 5.7 and Table 5.8. For the first set of tests, the simple automata reached nearly 85% of quality with at 10^3 order speedup. The parallel model, reached 100% of quality with the same speedup order of magnitude. As for the second set of tests, the simple automata reached just had just 32.93% of quality, while the parallel model, reached 99.29%.

5.5 DAC and ADC converters

On this last case study we analyze two blocks, a 5-bit digital-to-analog converter (DAC) built using a resistor ladder and a 5-bit flash analog-to-digital converter (ADC). The key functionalities of these elements is to interact with the inherently analog external

Test case	Time (s)		
	SPICE	Digital (simple)	Digital (parallel)
Synced reference	34.761	0.019	0.026
Out-of-sync reference	33.189	0.016	0.021

Table 5.7: PLL synchronizer basic 10^3 periods simulations speed comparison between an analog simulation and digital only simulation using ALIAS abstractions (simple and parallel).

Test case	Quality	
	Digital (simple)	Digital (parallel)
Synced reference	84.78%	100.00%
Out-of-sync reference	32.93%	99.29%

Table 5.8: Quality ratios for PLL synchronizer 10^3 periods simulation tests over ALIAS abstractions (simple and parallel).

world. While the DAC outputs *stimuli* data to such environment, the ADC and reads-in *stimuli* coming from that source. Figure 5.11a illustrates a such typical topology.

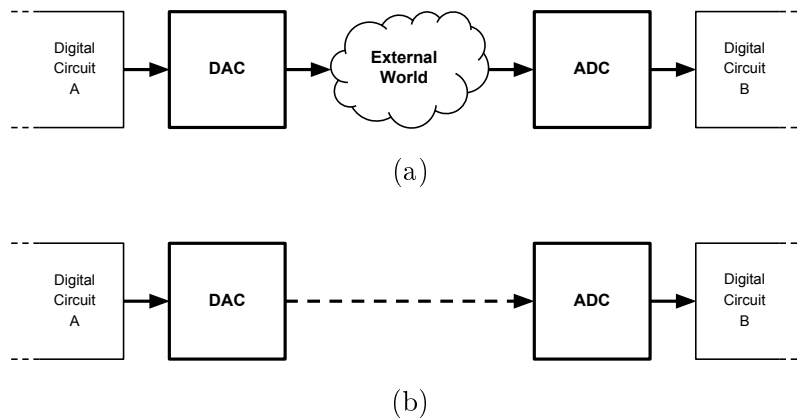


Figure 5.11: DAC and ADC (a) typical topology and (b) topology used for tests.

Thus, to test the behavior of such components from a digital perspective, we built an abstraction for each one of these blocks and then connected the DAC's outputs into the ADC's inputs, as shown in Figure 5.11b, covering all 5-bit conversion scenarios for each component. Both digital circuits at the layout ends (A and B) are simple interfaces, emitting stimuli that should be matched at the conversion end. With this topology in place, an analog simulation took $740.292s$ for a simple 1000 steps simulation, while the parallel abstracted ALIAS model powered one took with 100% of quality just $0.160s$, an $O(10^3)$ speedup. Due to the simple model non-determinism dominance, it only reached 3.71% of quality, running within $0.020s$. Differently from

the analog converter, into which the component operation could yield in just two possible outcomes, here we have 32 possible different results. Therein quality ratio for the simple model falls within the same ratio of a completely random system.

Chapter 6

Conclusions

We've developed and demonstrated ALIAS, a new tool for producing high quality abstract models from analog circuits based on existing data. As a result, digital engineers can now convert analog circuits into discrete models that scales linearly with the size of the simulation, in place of exponential analog simulations. Moreover, the abstractions can be created with adjustable settings for higher or lower precision marks, being synthesized with low user intervention and without the need of intensive SPICE simulations or complex mathematical methods, as present in current approaches.

6.1 Contributions

As a result of this work, digital engineers have a new tool for abstracting analog circuits into discrete models, which can be leveraged in the digital domain, providing abstractions that scales linearly with the size of the simulation, in place of exponential analog simulations. More, the abstractions can be created with adjustable settings for higher or lower precision settings.

For analog engineers, such abstractions can be used to prone for illegal behaviors spotted at higher levels, which could go unnoticed during the analog verification of a circuit, given the more complex nature of analog circuits. In addition, as digital engineers identify points which needs more coverage by the abstraction in the digital domain, analog engineers can further improve their tests, enabling a viable channel for testing not just the integration, but both analog and digital systems in a better way.

6.2 Limitations and Future Work

6.2.1 Automata history and transition improvements mechanisms

We aim to explore automata that takes into consideration the simulation history, thus avoiding past wrong choices and privileging the ones which did not lead the abstraction into error states. For this end, hardware prediction units, inspired in the concepts of branch prediction of modern processors [Hennessy and Patterson, 2011], can be used.

6.2.2 Segment Boundary Inference

Given the proposed mechanisms of selecting boundaries, we also target to explore better ways to infer the best candidate for boundary in a circuit. For that, we propose analyzing large trace collections and build a statistical profile of possible boundaries, reporting data such as how many occurrences and common strings within each boundary markers. After that, the user can either select the candidates that most looks like the actual resting state or opt for automated inference.

6.2.3 Recognition of Common Data Patterns

During our tests we often detected patterns on data signals that could be encoded and compressed down with smarter structures, such as counters, as is the case of the signal *out* of the integrator case study from Section 5.2. Therein, we expect to explore this field and employ established and or novel techniques on recognizing such patterns.

6.2.4 Data and control signals handling

We also plan to study a way of properly handling data and control signals going/coming from the analog domain, which could allow the construction of more compact and yet more precise abstractions. This idea can be applied to construct far more precise automata. As an example, verified with prototypes, the integrator from Section 5.2 reached 100% of quality.

6.2.5 Formal verification

Although we did not performed any tests with formal verification tools, which for RTL-based models there is still no free viable alternatives, all abstractions produced by ALIAS can be used on such systems.

Bibliography

- Aadithya, K. and Roychowdhury, J. (2012). Dae2fsm: Automatic generation of accurate discrete-time logical abstractions for continuous-time circuit dynamics. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 311–316. ISSN 0738-100X.
- Alexander, C. and Sadiku, M. (2008). *Fundamentals of Electric Circuits*. McGraw Hill Higher Education, 4th edition. ISBN 9780071284417.
- Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87--106. ISSN 0890-5401.
- Asarin, E., Dang, T., and Maler, O. (2001). d/dt: a verification tool for hybrid systems. In *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, volume 3, pages 2893–2898 vol.3.
- Barke, E., Grabowski, D., Graeb, H., Hedrich, L., Heinen, S., Popp, R., Steinhorst, S., and Wang, Y. (2009). Formal approaches to analog circuit verification. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09*, pages 724--729, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.
- Chen, J., Henrie, M., Mar, M., and Nizic, M. (2012). *Mixed-Signal Methodology Guide*. Cadence Design Systems, Incorporated. ISBN 9781300035206.
- El Tahawy, H., Rodriguez, D., Garcia-sabiro, S., and Mayol, J.-J. (1993). Vhdeldo: A new mixed mode simulation. In *Design Automation Conference, 1993, with EURO-VHDL '93. Proceedings EURO-DAC '93., European*, pages 546–551.
- Fairchild Semiconductor (2003). Mm74hc4046.
- Friedland, B. (2012). *Control System Design: An Introduction to State-Space Methods*. Dover Books on Electrical Engineering. Dover Publications. ISBN 9780486135113.

- Ghasemi, H. and Navabi, Z. (2005). An effective vhdl-ams simulation algorithm with event. In *VLSI Design, 2005. 18th International Conference on*, pages 762–767. ISSN 1063-9667.
- Hartong, W., Hedrich, L., and Barke, E. (2002). Model checking algorithms for analog verification. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 542–547. ISSN 0738-100X.
- Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition. ISBN 012383872X, 9780123838728.
- Henzinger, T. A., Ho, P.-H., and Wong-Toi, H. (1997). Hytech: A model checker for hybrid systems. In *Proceedings of the 9th International Conference on Computer Aided Verification, CAV '97*, pages 460–463, London, UK, UK. Springer-Verlag.
- Horowitz, P. and Hill, W. (2006). *The Art of Electronics*. Cambridge University Press. ISBN 9780521422284.
- IEEE (2001). IEEE Standard Verilog Hardware Description Language. *IEEE Std 1364-2001*, pages 324–343.
- IEEE (2013). Ieee standard for systemverilog–unified hardware design, specification, and verification language. pages 1–1315.
- Isaksen, B. and Bertacco, V. (2006). Verification through the principle of least astonishment. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 860–867. ISSN 1092-3152.
- Jaeger, R. C. (1987). *Introduction to Microelectronic Fabrication*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-14695-9.
- Karthik, A., Ray, S., Nuzzo, P., Mishchenko, A., Brayton, R., and Roychowdhury, J. (2014). Abcd-nl: Approximating continuous non-linear dynamical systems using purely boolean models for analog/mixed-signal verification. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 250–255.
- Karthik, A. and Roychowdhury, J. (2013). Abcd-l: Approximating continuous linear systems using boolean models. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–9. ISSN 0738-100X.

- Kularatna, N. and Kyung, C.-M. (2008). *Electronic circuit design: from concept to implementation*. Taylor & Francis, Hoboken, NJ.
- Kurshan, R. and McMillan, K. (1991). Analysis of digital circuits through symbolic reduction. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(11):1356–1371. ISSN 0278-0070.
- Liberali, V., Pettazzi, S., Ross, R., and Torelli, G. (2002). Challenges in mixed-signal integrated system design for industrial applications. In *Industrial Electronics, 2002. ISIE 2002. Proceedings of the 2002 IEEE International Symposium on*, volume 4, pages 1335–1340 vol.4.
- Linear Technology (1991). LT1011/LT1011A voltage comparator datasheet.
- Linear Technology (2015). Ltspice iv.
- Nagel, L. W. and Pederson, D. (1973). Spice (simulation program with integrated circuit emphasis). Technical report UCB/ERL M382, EECS Department, University of California, Berkeley.
- Nenzi, P. and Vogt, H. (2014). Ngspice Users Manual Version 26.
- Rashinkar, P., Paterson, P., and Singh, L. (2000). *System-on-a-chip Verification: Methodology and Techniques*. Kluwer Academic Publishers, Norwell, MA, USA. ISBN 0-7923-7279-4.
- Semiconductor Research Corporation (2008). SRC Research Needs in Computer-Aided Design and Test. Technical report.
- Shannon, C. (1998). Communication in the presence of noise. *Proceedings of the IEEE*, 86(2):447–457. ISSN 0018-9219.
- Silva, B. and Krogh, B. (2000). Formal verification of hybrid systems using checkmate: a case study. In *American Control Conference, 2000. Proceedings of the 2000*, volume 3, pages 1679–1683 vol.3. ISSN 0743-1619.
- Steinhorst, S. and Hedrich, L. (2008). Model checking of analog systems using an analog specification language. In *Design, Automation and Test in Europe, 2008. DATE '08*, pages 324–329.
- Stroud, C. E., Wang, L.-T. L.-T., and Chang, Y.-W. (2009). Chapter 1 Introduction. In Wang, L.-T., Chang, Y.-W., and Cheng, K.-T. T., editors, *Electronic Design*

- Automation: Synthesis, Verification, and Test (Systems on Silicon)*, chapter 1 - Introduction, pages 1--38. Morgan Kaufmann, Boston.
- Synopsys Inc. (2013). Verdi3 NPI Training – FSDB Model.
- Tuinenga, P. W. (1995). *SPICE (3rd Ed.): A Guide to Circuit Simulation and Analysis Using PSpice*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-158775-7.
- Vasudevan, S., Sheridan, D., Patel, S., Tchong, D., Tuohy, B., and Johnson, D. (2010). Goldmine: Automatic assertion generation using data mining and static analysis. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 626–629. ISSN 1530-1591.
- Waller, M. (2010). PSpice A/D Tips & Tricks and PSpice AA. *FlowCAD Application Note*.
- Williams, S. (2015). Icarus verilog.