Accurate Booleanization of Continuous Dynamics for Analog/Mixed-Signal Design

Karthik Aadithya

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2016-146
http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-146.html

August 12, 2016

Acknowledgement

To my mother, whose unconditional love and unwavering support have kept me going through thick and thin. To my father, who has never learned to stop worrying about me. And to my sister, for being there.

Accurate Booleanization of Continuous Dynamics for Analog/Mixed-Signal Design

by

Karthik Venkatraman Aadithya

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Jaijeet Roychowdhury, Chair
Professor Borivoje Nikolic
Professor Richard Karp
Professor Robert Brayton
Professor Allan Sly

Summer 2016

Accurate Booleanization of Continuous Dynamics for Analog/Mixed-Signal Design

Abstract

Accurate Booleanization of Continuous Dynamics for Analog/Mixed-Signal Design

by

Karthik Venkatraman Aadithya

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Jaijeet Roychowdhury, Chair

Analog/Mixed-signal (AMS) systems (*e.g.*, ADCs and DACs, charge pumps, comparators, SERDES systems and PLLs, *etc.*) are playing an increasingly important rôle in modern chip design: they have not only become key performance-limiting components in larger sub-systems, but they also now account for a disproportionately high share of design bugs (and associated designer-time and debugging costs). CAD tools for the accurate modelling, analysis, high-speed simulation, formal verification, and debugging of AMS systems have not been able to keep pace with the rapid growth in complexity of these designs. Therefore, effective, bug-free AMS design remains a largely unsolved problem today.

At the heart of this problem lies the fact that the analog components in AMS systems are modelled in a very different way from the digital components. While analog components are typically modelled as continuous dynamical systems using differential-algebraic equations (DAEs), the digital components are usually modelled using clean Boolean abstractions such as truth tables and finite state machines (FSMs). The formal analysis and verification of AMS systems, therefore, typically involves having to simultaneously reason about both continuous quantities and discrete quantities, which is inherently a very difficult mathematical problem.

A variety of "hybrid system" methods and frameworks have been developed to formally verify AMS designs. However, these methods tend to scale very poorly in the number of continuous (analog) variables involved, often being able to handle no more than 5 to 10 continuous variables. The enormous computational cost of adding extra continuous variables in turn forces the adoption of highly simplified behavioural macromodels for AMS components; these over-simplified models do not capture the behaviour of the underlying AMS components accurately.

In this dissertation, we propose a new approach to the problem of effective, bug-free AMS design. We call this approach "Booleanization", which involves *approximating* the continuous-domain behaviour of AMS components using purely Boolean models (such as FSMs). Booleanization works by first discretizing the voltages and currents in the given AMS design, and then encoding the analog dynamics of the given design in purely Boolean form using FSM state transitions. The finer the discretization of the circuit's voltages and

currents, the greater the accuracy with which the resulting Boolean model captures the circuit's continuous dynamics.

We propose three automated techniques for Booleanizing a wide variety of AMS systems and components: (1) DAE2FSM, which is based on an adaptation of Angluin's algorithm from computational learning theory, which works well for Booleanizing "digitalish" designs (*i.e.*, circuits whose intended functionality is purely digital, but which nevertheless exhibit significant performance-limiting analog traits and characteristics), (2) ABCD-L, a technique based on eigen-analysis that works well for Booleanizing Linear Time Invariant (LTI) continuous systems, and (3) ABCD-NL, a technique based on separating the underlying circuit's dynamics into DC and transient components, which works well for Booleanizing a large class of genuinely analog, genuinely non-linear AMS designs. These three approaches together form the ABCD[1] suite of Booleanization techniques.

Starting from SPICE-level netlists, we apply the techniques above to Booleanize a variety of AMS circuits including latches and flip-flops, digital logic blocks such as counters, charge pumps, delay lines, equalizers, filters, I/O links, ADCs and DACs, comparators, power grid networks, *etc.*. In each of these cases, we show that the Boolean models generated by ABCD are indeed able to capture the underlying analog dynamics of these systems with high accuracy.

We believe that Booleanization offers several compelling features. Firstly, since the generated models are all purely Boolean, they can be used in conjunction with existing state-of-the-art Boolean verification and model-checking engines (such as ABC) to formally verify AMS designs in a scalable way. Secondly, the Boolean models produced by ABCD can be simulated much more efficiently (in discrete-time, in the logical domain) than SPICE-level models; so, in many situations, ABCD-generated models may be usable as much faster, almost-as-accurate, drop-in replacements for SPICE. Thirdly, by virtue of being all-Boolean, ABCD-generated models enable one to accurately represent analog dynamics without incurring the severe penalty of adding continuous variables; so ABCD models may be usable in conjunction with existing hybrid system methodologies, frameworks, and verification flows – thereby helping these approaches scale to much larger problem sizes than they can do so at present. Finally, ABCD models can also be used in a variety of niche algorithms and "custom" analysis procedures aimed at solving targeted AMS design problems for specific systems and applications; for example, we have developed an efficient Boolean/LTI co-analysis procedure (called the Berkeley Eye Estimator, or BEE) for carrying out highly accurate worst case and stochastic eye diagram analysis of modern high-speed communication sub-systems that use correlated bitstreams and coding strategies to minimize bit error rates.

[1] Accurate Booleanization of Continuous Dynamics.

To my mother, whose unconditional love and unwavering support have kept me going through thick and thin. To my father, who has never learned to stop worrying about me. And to my sister, for being there.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to thank my advisor, Jaijeet Roychowdhury, for all the help, support, and encouragement that he has provided me with during the course of my PhD. Jaijeet has been an excellent teacher and a wonderful mentor to me all these years. He has not only contributed greatly to my research, but has also been a major factor in helping me grow as a person.

Next, I would like to thank my committee members, Professors Bob Brayton, Richard Karp, Bora Nikolic, and Allan Sly, who have all been very helpful to me and generous with their time.

I have also benefitted immensely from several of my collaborators. I would like to especially thank Alper Demir, Balaraman Ravindran, and Tomasz Michalak for their valuable advice and patient support that enabled me to become a better researcher.

I would be remiss if I did not mention Chenjie Gu, who was the first in our group to conceive of the idea of abstracting the behaviour of Differential-Algebraic Equation systems (DAEs) using Finite State Machines (FSMs). Indeed, it is fair to say that the initial work done by Chenjie along these lines towards the end of his PhD [1, 2] was a key factor that inspired me to choose Booleanization as my PhD research topic.

I also owe a debt of gratitude to my fellow grad students and colleagues at the DOP Center who have helped shape my research and improve it in innumerable ways. In particular, I would like to thank Baruch Sterin, who has always made himself available whenever I needed his inputs. I would also like to thank Tianshi Wang, Alan Mishchenko, Pierluigi Nuzzo, Sayak Ray, Palak Bhushan, Prateek Bhansali and Yu-Yun Dai for the many wonderful moments and conversations I have had with them over the years.

The internships that I did, and the subsequent interactions that I have had with the folks at Intel and at Sandia National Labs have been instrumental in shaping my thinking and approach to research. I am grateful to Wei-kai Shih, Ritochit Chakraborty, Eric Keiter, Joe Castro, Heidi Thornquist, Ting Mei, Scott Hutchinson, and Tom Russo for these enjoyable and thought-provoking interactions.

I would like to give a special note of thanks to Shirley Salanio, for her prompt and cheerful help every time I had to get administrative things done in the department.

Finally, I thank my family members, who have all been extremely supportive of my work. Special thanks go to my mom and dad, without whom I wouldn't even be in Berkeley today, and also to my little brother Dhruvesh who has been a source of pure joy since the day he was born.

Chapter 1

The Problem: Effective, Bug-free AMS Design

Effective and bug-free analog/mixed-signal (AMS) design is the main problem that this dissertation addresses.

This chapter introduces the problem, and discusses the features that make this problem an especially challenging one to solve. Also, to give the reader an appreciation of the importance and widespread nature of this problem (and the potential impact that an effective solution to this problem can have), this chapter also provides several motivating examples/instances of this problem drawn from a variety of application domains.

Finally, having presented the problem and the motivation for seeking solutions to it, this chapter provides an overview of the rest of this dissertation and the new approach to the problem contained therein.

1.1 Problem description

Virtually every electronic system, sub-system, and component manufactured today (*e.g.*, high-speed I/O links and communication sub-systems, Phase Locked Loops and sub-systems for Clock and Data Recovery, Serializer/Deserializer systems, microprocessor cores and SoCs, memories, *etc.*) contains a mixture of *analog* and *digital* circuitry. Such systems are called analog/mixed-signal (AMS) systems.

The analog components of an AMS system feature signals (voltages and currents) that vary continuously with time. Such components include, for example, amplifiers, filters, equalizers, *etc.*.

The digital components of an AMS system, on the other hand, feature signals that only take discrete values (typically, just two values 0 and 1). Examples of digital components include latches, flip-flops, logic gates, and any circuit that is built using combinational and sequential logic building blocks (adders, multipliers, encoders and decoders, application processors, digital signal processing (DSP) modules, *etc.*).

Consider a typical AMS system that consists of both analog and digital components. Clearly, for such a system to function correctly, the constituent analog and digital components need to be designed to appropriate sets of *well-defined specifications*, and these designs need to be *bug-free*.[1] Furthermore, the mixed-signal circuits that govern the *interaction* or the *interface* between the analog and the digital components (*e.g.*, the Analog to Digital converters (ADCs), the Digital to Analog Converters (DACs), the comparators and the sense amplifiers, *etc.*) all need to be designed to appropriate specifications that guarantee both correct functionality and adequate performance of the overall design.

Thus, since the analog, digital, and mixed-signal components all work together to realize the end-to-end functionality of the overall design (and also determine its overall performance), the specifications for these individual components, as well as their implementations, need to be bug-free.

Unfortunately, this is easier said than done, and therein lies the key problem faced by AMS designers: it is notoriously difficult to design a correctly functioning (non-trivial) AMS system that meets every one of its design specifications (functionality-wise and performance-wise).

The next section discusses the increasing importance of AMS components in modern chip design, and it also describes in detail why AMS design is such a challenging problem.

1.2 Key features/aspects of the AMS design problem

1.2.1 The increasing importance of AMS systems

AMS systems are becoming increasingly important in chip design. This is because, for performance reasons, a variety of components (*e.g.*, the USB interface, I/O, memory, *etc.*) in systems like microprocessors and SoCs that traditionally used to be implemented using predominantly digital blocks are now increasingly being implemented using AMS blocks instead [3, 4]. Indeed, with the advent of sub-45nm CMOS technologies, it has now become more cost-effective than ever (and also more advantageous from a performance-per-Watt standpoint) to integrate analog/RF functionality into SoCs [3, 4].

For example, Fig. 1.1 depicts the number of unique analog and analog/mixed-signal circuits in Intel® microprocessors ranging from 250nm down to the 32nm node. The steadily increasing number of unique AMS circuits in recent technologies indicates a significant expansion in the types of functionality that are now realized using AMS components in modern chipsets.

Also, the last few generations of microprocessor/SoC chipsets from prominent companies like Apple, Intel, Qualcomm, *etc.*, indicate a pronounced trend towards ever tighter inte-

[1] To be precise, even if the constituent components do not strictly adhere to their specifications, the overall system may still function correctly if these discrepancies are within a margin of tolerance or if the effects of these discrepancies are not observable from an end-to-end input/output or performance perspective (*e.g.*, if a "bug" only affects false paths in a digital design). However, this is a technicality, and we will assume that the definition of "bug-free" already subsumes this.

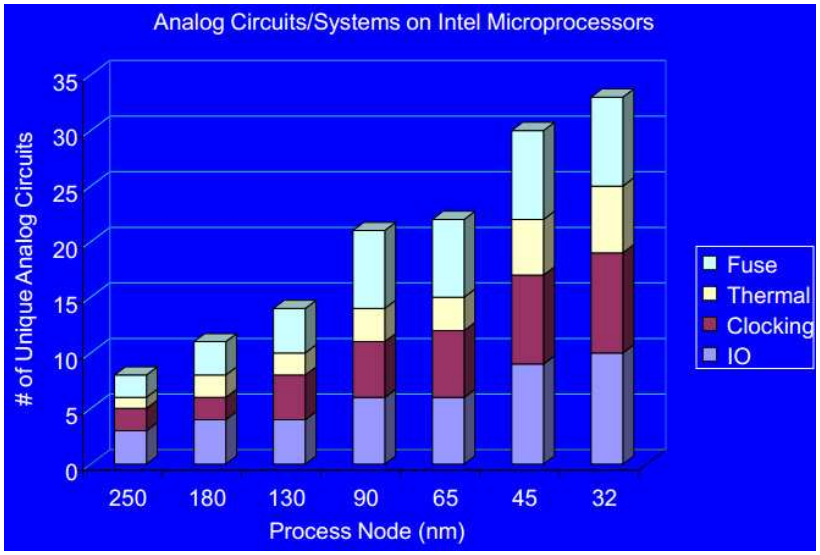


Figure 1.1: The number of unique analog and AMS systems in microprocessors (which have traditionally been predominantly digital) has been growing at a fast clip in recent times. Figure credits: Rachel Parker (Intel).

gration between the analog, the digital, and the mixed-signal components on an SoC. This also foreshadows an increasingly prominent and important rôle for AMS components going forward.

1.2.2 Disproportionate share of bugs, resources

Although AMS components typically make up only a relatively small percentage of the overall transistor count in modern chips such as SoCs, these components have now become key *bottlenecks* that have an *outsize impact on important system-level performance metrics* such as data-rate/throughput, power consumption, noise margins, *etc.* [3, 5].

In addition to their increasingly significant rôle from a functionality and performance perspective, AMS components also now account for a significant proportion of design bugs, designer time, and debugging cost.

Indeed, according to a recent whitepaper[2] circulated by Intel, AMS components now account for approximately 20% of all design bugs over the design cycle of modern microprocessors. This is very disproportionate considering that the transistor count of the AMS components is almost insignificant relative to that of the overall microprocessor (which typically has billions of transistors devoted to purely digital circuitry). Also, because these bugs are typically located at the interface between analog and digital components (and can cause failures even when the respective analog and digital components each satisfy their individual

[2]We are not allowed to cite this whitepaper because it was shared with us in confidence, and it is not in the public domain.

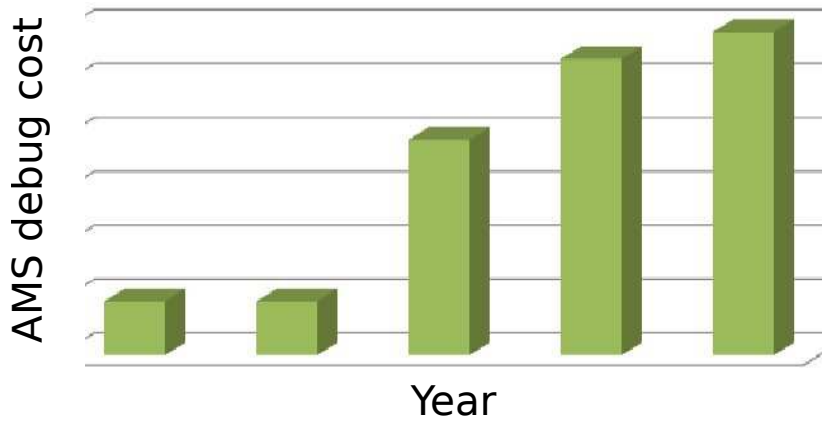


Figure 1.2: The cost of debugging AMS components in microprocessors has been increasing rapidly in recent times. Figure credits: Kalpana Kothapally and Tom Lertpanyavit (Intel).

specifications), they tend to be subtle and difficult to isolate, reproduce, and fix. For example, Fig. 1.2 shows that the cost of debugging AMS components in Intel® microprocessors has been sharply increasing in recent years.[3] This reflects a disproportionate share of designer time and other resources being devoted to the AMS components of the chip, which raises design and debugging costs for these components significantly. Furthermore, in spite of the expenditure of such disproportionate resources, AMS components continue to remain some of the more unreliable and "brittle" portions of the overall design, accounting for numerous respins, increased time to market, and lingering bugs that cause failures when the design is deployed in the real world.

1.2.3 Analog models vs Digital models

At the core of any AMS design, we have a set of analog components interacting with a set of digital components. Chip designers typically use a variety of CAD[4]/EDA[5] tools and software for modelling, analyzing, simulating, verifying, and debugging these components. For example, SPICE is a CAD tool used by chip designers to simulate the analog components, while ABC is a CAD tool used by them for verifying the digital components.

Internally, these software tools use very different kinds of *mathematical abstractions* to represent the behaviour of the underlying circuits and components. For example, SPICE represents the behaviour of the underlying analog circuit using the mathematical abstraction of a *Differential-Algebraic Equation* system (or DAE), while ABC uses the mathematical abstraction of a *Finite State Machine* (or FSM) to represent the behaviour of the underlying digital component or circuit.

[3]For confidentiality reasons, Intel did not release actual numbers for the data in this plot.
[4]Computer Aided Design.
[5]Electronic Design Automation.

This is true in general: the *mathematical language* used to describe the analog components of a design is starkly different from that used to describe the digital components. This is a fundamental feature arising from the inherently different natures of the analog and the digital components. The analog components feature continuously varying voltages and currents, so the mathematical language used to model them needs to have the expressive power to represent continuously varying quantities (but it doesn't need to be able to represent discrete quantities since those never arise in an analog setting). The language of differential and algebraic equations is thus an ideal choice for modelling the analog components. On the other hand, the digital components feature signals that can take on only discrete values (and evolve only in discrete time), so the mathematics of automata theory and FSMs is ideal for the digital components.

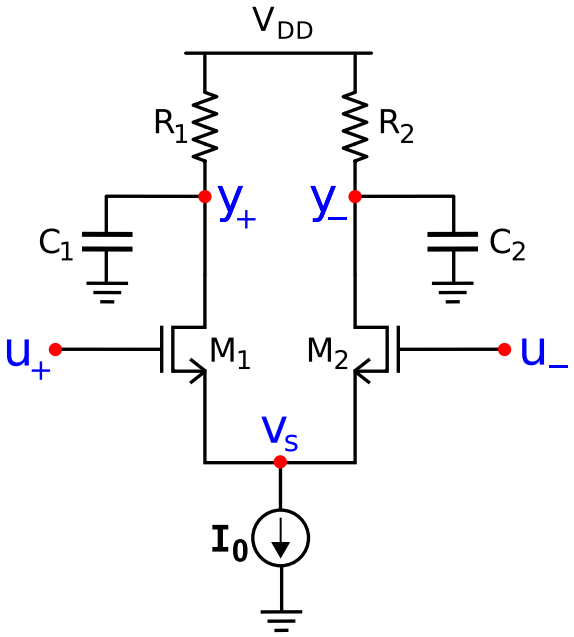


Figure 1.3: A differential pair circuit (a common building block used in analog components).

For example, Fig. 1.3 shows a differential pair, a common analog circuit used as a building block in many analog components. The circuit accepts 2 (time-varying) inputs, denoted $u_+(t)$ and $u_-(t)$, in addition to the supply voltage V_{DD} and the bias current I_0. Given these inputs, the circuit produces two outputs $y_+(t)$ and $y_-(t)$. Because this is an analog circuit, all its voltages and currents vary continuously with time. The behaviour of this circuit (*i.e.*, the equations governing how the outputs are produced from the inputs) is represented using the following differential-algebraic equation system:[6]

[6]In this system of equations, the currents flowing through the transistors M_1 and M_2 are assumed to be instantaneous functions of the drain and gate voltages across the respective transistors. For the purposes of this example, we ignore effects like internal nodes, parasitic capacitances, *etc.*.

$$\frac{d}{dt}\begin{bmatrix} C_1\ y_+(t) \\ C_2\ y_-(t) \\ 0 \end{bmatrix} + \begin{bmatrix} \frac{y_+(t)-V_{\mathrm{DD}}}{R_1} + I_{M_1}(u_+(t)-v_s(t),\ y_+(t)-v_s(t)) \\ \frac{y_-(t)-V_{\mathrm{DD}}}{R_2} + I_{M_2}(u_-(t)-v_s(t),\ y_-(t)-v_s(t)) \\ I_{M_1}(u_+(t)-v_s(t),\ y_+(t)-v_s(t)) + I_{M_2}(u_-(t)-v_s(t),\ y_-(t)-v_s(t)) - I_0 \end{bmatrix} = \vec{0}. \tag{1.1}$$

If, for example, the circuit above is simulated using an analog CAD tool like SPICE, the underlying mathematical model that SPICE will construct for the circuit will correspond to the DAE system above.

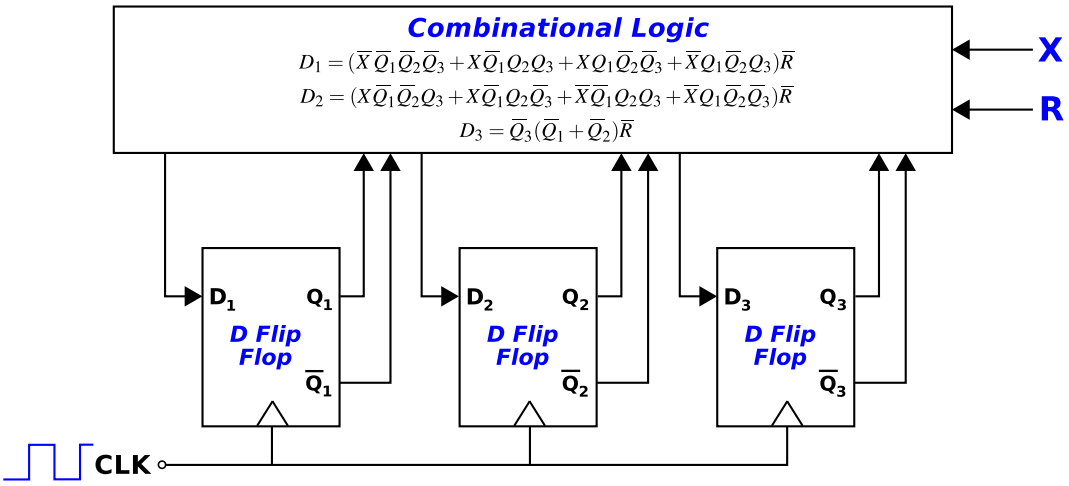


Figure 1.4: Schematic of a 0-to-5 increment/decrement counter with reset (a common building block used in digital components).

At the other end of the spectrum are the digital components. For example, Fig. 1.4 shows a digital counter, a typical building block used in many digital systems. The particular implementation showed in Fig. 1.4 is a simple circuit capable of counting up and down between 0 and 5. The system takes two *digital* inputs (`X` and `R`) in addition to a *clock signal* `CLK`. Given these inputs, the system produces 3 digital outputs (Q_1 through Q_3). Because this is a digital system, the signals `X`, `R`, Q_1, Q_2, and Q_3 are all *discrete bits*: they are either 0 or 1 at any given time, and they change only at the positive edge of each clock cycle (*i.e.*, when `CLK` transitions from 0 to 1). The input `R` decides whether the counter should reset itself or not: if `R` is 1, the counter resets itself to 0 in the next clock cycle, and if `R` is 0, the counter increments or decrements its state between 0 and 5 depending on whether the input `X` is 1 or 0 respectively. The equations for D_1, D_2, and D_3 (which are the values that Q_1, Q_2, and Q_3 will assume at the next clock cycle) are implemented using logic gates (*e.g.*, AND, OR, NOT, NAND, XOR, *etc.*), and are shown within the combinational logic block in the schematic above.

The behaviour of the counter above can be represented using the FSM shown in Fig. 1.5. This FSM is a *Mealy machine*: at any given time, the system is in one of 6 possible states

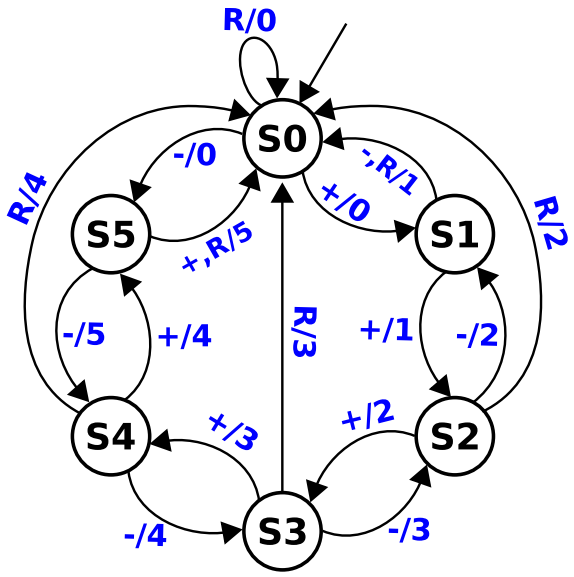


Figure 1.5: An FSM model for the counter in Fig. 1.4.

S_0 through S_5; these represent the states 0 through 5 of the counter respectively. At every positive clock edge, the system looks at the input (the bits `X` and `R`) and decides on an appropriate *action* to take. This action could be either a + (incrementing the count), or a − (decrementing the count), or an R (resetting the count to 0). Depending on the current state, and the action chosen, the system *transitions* to a new state and produces a discrete output. These transitions are represented by the arcs in the FSM shown in Fig. 1.5. Each arc is annotated with the action performed and the output produced (with output 0 standing for the bits 000, the output 1 standing for the bits 001, *etc.*).

If, for example, the circuit above is described to a digital CAD tool like ABC [6], the underlying mathematical model that ABC will construct for the circuit will correspond to the FSM above.

The two examples above (the differential pair and the counter) highlight the key differences in the mathematical models used to represent analog and digital components respectively. The analog mathematical models are systems of differential and algebraic equations that can only represent continuously varying quantities (not discrete ones), whereas the digital mathematical models are finite state machines that can only handle discrete quantities (not continuous ones).

1.2.4 Practical difficulties confronting AMS design and analysis

The *fundamental incompatibility* between the mathematics of continuous systems and that of discrete systems (illustrated by the examples in the previous section) is the main reason why the modelling and analysis of AMS designs (which involve a mixture of both kinds of

systems) is such a challenging problem: the models used for analog designs do not mix well with the models used for digital designs.

In practical terms, software tools for analog design (such as SPICE) that rely on continuous mathematics do not work for purely digital or mixed-signal components. Similarly, the most powerful and capable software tools that we have for digital design (such as ABC) do not work for purely analog or mixed-signal components.

Indeed, a wide variety of tools, techniques, algorithms, analysis procedures, and heuristics have been developed for designing analog components. These include, for example, several techniques for DC analysis (the Newton-Raphson algorithm, homotopy, initialization and limiting heuristics, *etc.*), linearization and AC analysis, linear multi-step methods (*e.g.* , Forward Euler, Backward Euler, Trapezoidal integration, the Gear methods, *etc.*) for transient analysis, shooting, harmonic balance, and other methods for the analysis of periodic systems, several algorithms for noise analysis, eye diagram analysis, sensitivity analysis, *etc.*, as well as a variety of techniques for the optimization and synthesis of analog circuits [7–9]. But all these algorithms and techniques assume that the underlying system is available as a DAE. Therefore, while these methods are useful for analyzing and designing analog components (often containing millions of transistors), they do not apply to digital or mixed-signal systems.

Similarly, a vast body of work and literature is available for the analysis and verification of purely digital systems [6, 10, 11]. In many ways, CAD tools and algorithms for digital systems (*e.g.*, the rapid advancements made in the design and implementation of SAT solvers, BDD/AIG based Bounded Model Checking (BMC) tools, Property Directed Reachability (PDR), several Satisfiability Modulo Theories (SMT) and solvers, *etc.*) are even more impressive than those available for analog systems. For example, with the advent of symbolic model checking, tools like ABC are now routinely used in the industry for the sequential equivalence checking of very large digital designs (often containing millions of logic gates and several thousand flip-flops, with the number of possible states far exceeding the number of atoms in the known universe). But again, all these tools and techniques assume that the underlying system is represented as an FSM. While this assumption holds true for digital designs, it breaks down for analog or mixed-signal systems, thereby rendering this vast body of work practically irrelevant for AMS design.

Thus, it is the mathematical difficulties that stand in the way of combined simultaneous analysis of analog and digital systems that are chiefly responsible for the slow progress in developing truly effective CAD tools for AMS design. To be sure, much effort has been invested in the development of CAD tools and algorithms for AMS design. The approaches tried so far fall mainly under the umbrella of "hybrid system" research, *i.e.*, the analysis of systems that contain a mixture of continuous and discrete quantities. Indeed, an overview of these approaches is presented in Chapter 2. However, these approaches often suffer from lack of modelling accuracy/fidelity, as well as scalability limitations, and are therefore not yet well-suited for the design of practical real-world AMS systems. In other words, the pace of development of new CAD tools, techniques, and algorithms for AMS design (including tools for modelling, analysis, simulation, verification, testing, and debugging of AMS designs) has

thus far not been able to catch up with the rapid growth in the size and complexity of these systems. This dissertation is an attempt towards bridging this gap by developing a new approach (called Booleanization, introduced in Chapter 3) to the AMS design problem.

1.2.5 Sub-problems of the AMS design problem

The topic of AMS design is a rather large one with a wide scope, so it is often beneficial to think in terms of smaller, more well-defined problems that arise frequently in practice. This section breaks down the overall AMS design problem, and presents it in terms of a set of smaller, better-focused sub-problems. These include AMS modelling, AMS analysis (which in turn comprises intuitive understanding, simulation, verification, and custom analyses), AMS debugging, and AMS testing.

Due to the rôle played by AMS components in determining system-level performance and due to the increased incidence of design bugs associated with AMS components, the sub-problems discussed below (such as AMS modelling, analysis, simulation, verification, debugging, test, *etc.*) are key challenges confronting the EDA industry today.

1.2.5.1 AMS modelling

One of the first steps in AMS design is *modelling*. This means developing mathematical abstractions for the behaviour of the underlying AMS design and its various components.

As described in Section 1.2.3, the analog components in an AMS design can usually be modelled as a system of differential-algebraic equations, the digital components can be modelled as finite state machines, and the mixed-signal components can usually be modelled using a "hybrid system" approach. Below, we discuss the two most important factors to be considered while developing and using such abstractions to model the behaviour of AMS designs: (1) the model's *accuracy*, and (2) the model's *amenability to analysis*:

Accuracy: Clearly, if we use a mathematical model to represent and reason about a real-world AMS component, we should ensure that the predictions made by the model are closely in line with the actual behaviour of the AMS component under all operating conditions. This is the *accuracy* issue. For example, if we use a SPICE netlist (which, at its core, is a DAE system as described in Section 1.2.3) to model an analog component in our AMS system, we should take care to choose an appropriate transistor model (*e.g.*, Shichman-Hodges vs BSIM vs PSP), an appropriate set of parameters (*e.g.*, transistor widths and lengths, threshold voltages, parasitic effects, *etc.*), *etc.*, so that our SPICE simulations of the analog component accurately reflect its real-world behaviour. To take another example, even the so-called "purely digital" building blocks (such as latches, flip-flops, logic gates, *etc.*) in today's deeply-scaled technologies often exhibit significant analog characteristics (*e.g.*, non-ideal waveform shapes, inter-symbol interference, crosstalk, overshoot/undershoot, dispersion and distortion, increased effect of parasitics, *etc.*) that have a meaningful impact on system-level performance – so, for such components, a DAE-based model (such as a SPICE

model) is often more appropriate than a traditional all-digital ideal FSM model (even though the component's *intended* functionality is purely digital).

A quick rule of thumb in this context is the (rather imprecise) "eyeball metric" for typical AMS verification applications: when viewed with the naked eye (without magnification, and at a reasonable reading distance), plots of the outputs produced by the AMS model being used by the verification tool should resemble plots produced by SPICE simulation, on all relevant inputs. Of course, like all rough guidelines, this should be followed with caution, and some applications (like the verification of certain RF systems) may require much higher accuracy than what satisfies the eyeball metric.

Amenability to analysis: This has to do with how we plan to *use* the developed AMS model. For example, if we want to use the model for *timing analysis*, we should make sure that (a) the model captures the underlying system's delays and other timing-related properties accurately, and (b) our timing analysis tool is able to efficiently make use of the model.

Unfortunately, the two factors above (accuracy and amenability to analysis) are often in conflict: highly accurate models tend to be least amenable to analysis and vice-versa. Indeed, to a large degree, our choice of model is constrained by the analysis we wish to carry out using the model. Specifically, if our analysis engine does not accept a particular kind of model, we have no choice but to use a different (possibly less accurate) model. For example, if our analysis engine is ABC, we cannot use a differential equation system based model (such as a SPICE model), because ABC can only work with Boolean models (such as FSMs). Furthermore, even if our analysis engine accepts our model, *scalability* is often an important consideration: for example, if our analysis engine is a hybrid system verification tool (such as the LEMA toolkit developed by Chris Myers *et. al.*), we often cannot use it to verify systems that have hundreds (or sometimes even tens) of continuous quantities because the core reachability analysis algorithms available for such kinds of systems scale very poorly with the number of continuous variables, and will be computationally infeasible for such models.

Indeed, many hybrid system approaches tend to adopt overly simplistic models for the underlying AMS components chiefly because the underlying analysis algorithms scale poorly (please see Chapter 2 for a more detailed discussion). But this can also be dangerous because such over-simplified models often ignore important aspects of the underlying system's behaviour as it operates in the real-world; therefore, analyses based on such models may prove untrustworthy or of little relevance.

At the other extreme are the SPICE-level models (especially those that use advanced transistor models such as BSIM or PSP). These are usually highly detailed, and are able to accurately predict the behaviour of the underlying digital, analog, or mixed-signal component. However, these models tend to be very limited from an "amenability to analysis" standpoint: outside of SPICE simulations (which are also often unacceptably slow, especially for large AMS designs), one usually cannot use SPICE models for any other kind of analysis. For example, one cannot usually carry out formal verification using SPICE-level models, because these models often represent large and complicated systems of differential-algebraic equations

for which there are no known analytical solutions/procedures, and therefore the associated verification problems tend to be extremely hard (in many cases provably computationally equivalent to the halting problem [12]) and well outside the scope of any existing verification tool or technique.

1.2.5.2 AMS analysis

Once we have a mathematical model for our AMS system, the next step usually involves carrying out some kind of *analysis* using this model. Many kinds of analysis problems frequently arise in AMS design. This section provides simple examples/instances of these problems below. For more concrete examples with greater relevance to industry-scale AMS designs, please see Section 1.3.

Intuitive understanding: This is not a well-defined analysis, but the goal here is for the AMS designer to simply gain some intuition about the underlying design, which is often used to improve the design. For example, if the designer wishes to understand the performance characteristics of an I/O link, a quick way to gain intuition might be to linearize the link and plot its transfer function. Even though the result may be highly approximate, it can sometimes provide significant insights to the designer (helping him decide, for instance, the kind of equalization circuitry to use with the link).

High-speed simulation: Very often, an important component of AMS design involves simulating the design for a wide variety of input waveforms, parameters, operating conditions, *etc.*. Again, there is a tradeoff between simulation speed and accuracy. Using complicated differential-algebraic equation based models (*e.g.*, SPICE models that use BSIM transistors), at least for the analog components of the design, often yields the most accurate results, but can sometimes be unacceptably slow. Using reduced complexity macromodels (*e.g.*, simpler transistors like Shichman-Hodges, or hybrid behavioural models for certain components) can often speed up AMS simulation by orders of magnitude, but accuracy is likely to suffer.

Formal verification: The formal verification problem for AMS designs is stated as follows. Given an AMS design (expressed as a mathematical model such as a SPICE model or an FSM or a hybrid system model), and given a *property* that the AMS design is supposed to satisfy (usually this property pertains to either the design's functionality or its performance), the problem is to determine (in a fully automated, *formal* way, with minimum manual intervention) whether the given model for the AMS design satisfies the given property or not. If the model indeed satisfies the property, the AMS verification tool is expected to produce a *proof* of this fact. If not, it is expected to produce a *counter-example* demonstrating that the model does not, in fact, satisfy the property (the most common counter-example being a sequence of input waveforms to the model that result in the property being violated).[7]

[7]In this dissertation, we use the term *verification* in the same way it is used by the digital CAD community, referring to the problem of checking whether a given *mathematical model* satisfies a given *property*. The mathematical models encountered in digital design are frequently amenable to this kind of *model checking*, making it possible to mathematically prove system-level properties. In the analog CAD community, however,

For example, let us say we have a hybrid model for an 8-bit ADC that accepts inputs between 0 Volts and 2 Volts. In this case, our property might be the following: "as long as the input to the ADC eventually settles to a constant value that is greater than 1.5 Volts, the most significant bit in the ADC output always settles to a 1 within 50 nanoseconds of the input settling.". Note that *proving* this property for the ADC is *not* the same as simply running a simulation. This is because the property should be proved for *all initial states* of the ADC, and for *all input waveforms* that eventually settle at 1.5 Volts or higher, and there are an infinite number of such cases to consider. Therefore, a proof of this property has to be obtained by *formally analyzing* the underlying ADC model, not just simulating it for a specific input waveform.

Therefore, AMS verification is inherently a very difficult problem to solve. If it is solvable at all, it is often NP-Hard, and existing solutions tend to be computationally infeasible for practical real-world AMS designs.[8] However, as the underlying model for the AMS design is made simpler, the problem tends to become more tractable. This is why most existing formal/semi-formal approaches to AMS verification make a number of simplifying assumptions while modelling the underlying AMS design for verification. In particular, most existing AMS verification approaches only work with simplified behavioural models for the underlying AMS components; these models often consist of only one or two continuous signals (as opposed to the corresponding SPICE model which may easily contain hundreds or even thousands of continuous signals), and the dynamics exhibited by these models also often tend to be highly simplified approximations to the real design (for example, non-linear dynamics is often modelled by piecewise, constant-slope approximations, *etc.*). Thus, the device/component models used for AMS verification are often much less detailed, and of lower accuracy, compared to the models typically used for AMS simulation. While the former are usually hybrid behavioural models with just one or two continuous quantities, the latter are often detailed SPICE-level models that offer a much higher degree of accuracy. This is another example of the tradeoff between accuracy and amenability to analysis (as discussed in Section 1.2.5.1).

Notwithstanding its inherent difficulty, the AMS verification problem is of prime importance because, clearly, the ability to formally prove properties of AMS designs (or to quickly find counter-examples thereof) has huge implications for bug-free AMS design. Therefore, scalable AMS verification is an especially important unsolved problem gaining traction in the EDA industry today. And while much research effort has been invested in the search

it is more common to refer to this problem by the somewhat weaker term *validation* (instead of *verification*), because the mathematical models encountered in analog design, such as DAEs, are frequently not directly amenable to formal reasoning, model checking, or proof generation, which in practice usually means that the best way to check properties is to run a large number of simulations. While this may give the designer confidence that the system satisfies desired properties, it does not usually constitute a mathematical guarantee (unlike the proofs generated by digital verification tools), making the term validation more appropriate in this context than verification.

[8] Indeed, depending on the underlying model for the AMS design, the problem as stated may even be undecidable, *i.e.*, provably computationally equivalent to the halting problem [12].

for solutions to this problem, the state-of-the-art needs to be advanced signigicantly before practical AMS verification for industry-scale designs becomes a reality.

Here, we would like to draw an important distinction between AMS verification and traditional Boolean verification. The verification problem described above is well-known in the context of purely digital/Boolean designs, *i.e.*, designs that only contain purely digital components (*e.g.*, logic gates, latches, flip-flops, and other combinational and sequential logic elements). The key difference between Boolean verification (the verification of purely digital designs) and AMS verification (the verification of designs containing both analog and digital components) is that Boolean verification is much more scalable. Indeed, Boolean verification is a mature, well-developed, and longstanding field of research where outstanding progress has been made over the last few decades – to the point that a variety of tools, techniques, and algorithms that can comfortably verify even very large real-world (digital) designs are readily available today. Indeed, with the advent of high-speed SAT solvers, symbolic model checking, BDDs (Binary Decision Diagrams), the increased use of AIGs (And Inverter Graphs) and their analysis via sophisticated techniques for bounded model checking, property directed reachability, and a variety of special-purpose satisfiability modulo theories and solvers, the state-of-the-art Boolean verification tools (*e.g.*, ABC [6]) for purely digital systems have become so advanced that they are routinely used in the industry to verify designs with millions of gates and several hundred flip-flops. By contrast, AMS verification is far less developed as a field of research, and the tools and techniques available today for AMS verification are not quite powerful enough for verifying real-world AMS designs.

The chief feature that makes AMS verification more challenging than Boolean verification is that AMS designs contain analog signals. This means that the AMS verification tool has to be able to reason about quantities that can take on a continuous range of values. By contrast, in Boolean verification, there are only 2 possible values for each circuit signal, either 0 or 1. The need to represent and reason about continuous quantities often creates an exponentially worse state-space explosion problem relative to the state spaces of purely Boolean systems, and the associated computational costs are often enormous, rendering AMS verification techniques orders of magnitude slower than their Boolean counterparts. This is why Boolean verification techniques are able to work with designs containing millions of logic gates, but even state-of-the-art AMS verification techniques are often unable to verify systems with more than a few (*e.g.*, 5 to 10) continuous signals.

The core ideas that we propose in this dissertation (Chapter 3) take advantage of the fact that Boolean verification can be much faster than AMS verification. Indeed, a key contribution of this dissertation is to show how AMS systems can often be modelled using purely Boolean approximations (via a process called Booleanization, covered in Chapter 3), thereby making it possible to leverage state-of-the-art Boolean verification tools, and apply them to AMS verification.

Custom analyses (*e.g.*, eye diagram analysis): In addition to problems such as high-speed simulation and formal verification, AMS designers are often confronted with a number of "niche" problems that require developing "custom" solutions, algorithms, and analysis procedures. Many of these "custom analysis" problems tend to be design specific.

For example, eye diagram analysis is one such problem. It arises almost exclusively in the context of high-speed links and other communication sub-systems. It has important applications in signal integrity analysis, bit-error rate prediction, *etc.*, but it qualifies as a "niche" problem because it is not as well-known as, and does not have as wide a scope as, for example, AMS verification. AMS design is full of such problems requiring custom solutions, and large chip design houses such as Intel often invest heavily in the development of custom EDA tools and analysis procedures specifically for such problems. The AMS models used for such kinds of analyses are often the same as those used for more mainstream analyses. Also, the scalability of these custom analysis procedures may depend significantly on the model chosen, so the tradeoff between accuracy and amenability to analysis is present here as well.

1.2.5.3 AMS debugging

AMS debugging is the problem that logically follows AMS analysis/verification. Assuming that the analysis/verification procedure has found a bug in the underlying AMS design, the AMS debugging problem has to do with isolating the bug, understanding its root causes, and eventually, fixing it. While AMS debugging has not received as much attention from the research community as AMS verification, it is nevertheless an important problem that must be addressed in order to enable effective, bug-free AMS design.

1.2.5.4 AMS testing

AMS testing refers to the problem of generating test input stimuli for manufactured AMS designs, to help distinguish correctly functioning chips from failing ones. As with formal verification, much work has been done (with highly impressive results) to enable scalable test generation for purely digital designs. However, for AMS designs, progress towards solving this problem has been significantly slower.

1.3 Motivating examples/instances of the AMS design problem

Below, we present some examples, drawn from a variety of application domains, that illustrate the importance and key features of the AMS design problem in the context of the previous sections of this chapter.

1.3.1 A SAR-ADC

This first example discusses the problem of bug-free AMS design in the context of a successive approximation analog to digital converter (or SAR-ADC). Fig. 1.6 shows a high-level schematic of this system.

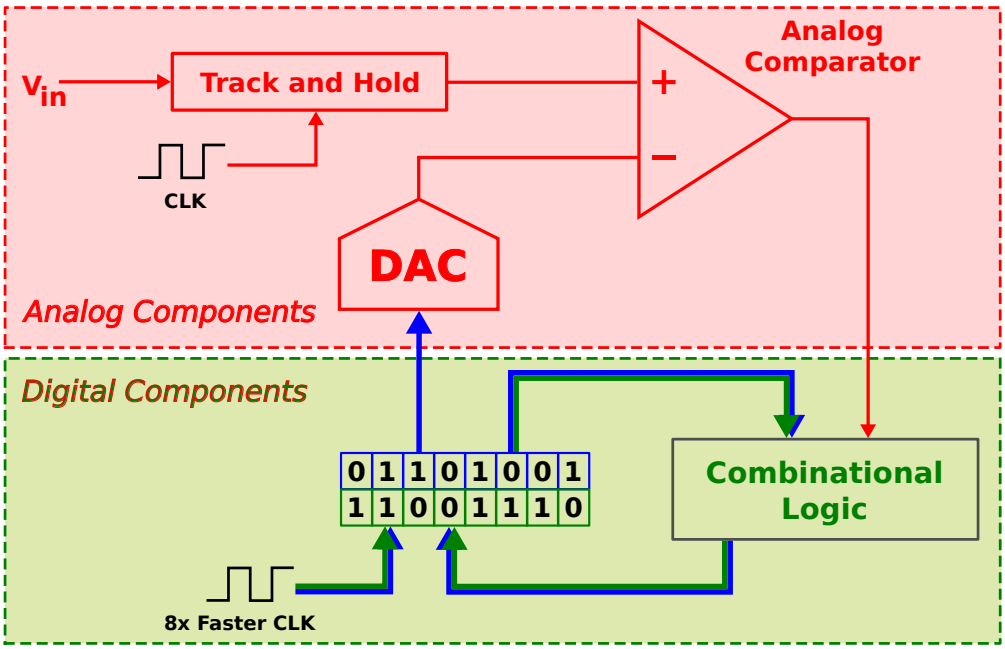


Figure 1.6: Schematic for a successive approximation ADC (or SAR-ADC).

As the figure shows, this system is a canonical mixed-signal system, *i.e.*, it contains a mixture of several interoperating analog and digital components that are crucial to realizing the overall system's core "ADC" functionality.

The analog/mixed-signal parts of the system consist of a "track and hold" unit, an analog comparator, and a digital to analog converter (DAC), as indicated in Fig. 1.6.

The digital components include several registers, as well as significant amounts of combinational logic (Fig. 1.6).

The system works as follows: the overall functionality of the system is as an analog to digital converter. That is, given an analog waveform (the input voltage V_{in}), the system quantizes it into a vector of bits. These bits are stored (and can be read from) the *registers* in the system (shown in Fig. 1.6), which are implemented as flip-flops that constitute a portion of the digital part of the system.

Initially, all the output bits are treated as unknown (they need to be set by the ADC's logic). The ADC accomplishes its function by iterating through a sequence of steps, determining one output bit (the most significant output bit that has not yet been determined) at each step. This is similar to the classic *binary search* algorithm, and in fact, the entire SAR-ADC can be viewed as a hadware implementation of binary search.

In general, because the input waveform V_{in} keeps changing with time, it is not guaranteed to remain constant while the SAR-ADC executes its many iterations. This is where the "track and hold" unit comes in: it is the "track and hold" unit's responsibility to *sample* the input waveform from time to time, and hold this (analog) sample at a constant voltage

value, irrespective of any changes to V_{in} after the sample has been recorded, so that the rest of the ADC can execute its binary search algorithm with a steady input sample.

The algorithm itself is relatively straightforward. At each iteration, a certain number of most significant bits (MSBs) are known, and the next MSB (whether 0 or 1) needs to be determined. This determination proceeds as follows: the next MSB is determined to be 0 if the input sample held by the "track and hold" unit is *lower* than a *threshold voltage* (which is determined from the MSBs that are already known), and it is determined to be 1 if the input sample is higher than this threshold.

For example, let us say we have an 8-bit ADC that operates between 0 Volts and 1 Volt, and let us say that three of the MSBs are already known to be 101. Thus, the output of the ADC will be of the form $101XXXXX$, where X represents an unknown bit. Now, the fourth MSB can be determined as follows. Since this is an 8-bit ADC, where 00000000 represents 0 Volts and 11111111 represents 1 Volt, we know that if the first 3 MSBs are 101, then the ADC output is limited to the range 10100000 to 10111111. The former translates to an analog value of about 0.63 Volts on our scale, while the latter translates to an analog value of about 0.75 Volts. Splitting the difference evenly, we may choose our threshold voltage to be around 0.69 Volts, *i.e.*, if the input sample is higher than 0.69 Volts, we may choose to set the fourth MSB to 1, and otherwise we may set it to 0.

Note that the above binary search logic for determining the threshold voltage requires a conversion from the bit vector scale to the analog scale: this is precisely what the DAC in the above schematic does. For instance, in the example above, the ADC is trying to determine the fourth MSB while its output eventually settles to a value in the range 10100000 to 10111111. So, in this iteration, the DAC may be fed the input 10110000 or 10101111 (akin to the mid-value threshold of 0.69 Volts above). The output voltage of the DAC in response to this input becomes the threshold voltage against which the "track and hold" output is compared. Such voltage comparisons are done by an analog comparator (Fig. 1.6), which, in each ADC iteration, returns a single bit that is used by the combinational logic to set the next higher MSB. This process continues until all the output bits have been set, at which time the ADC output is returned and a new input sample is recorded by the "track and hold" unit, thereby beginning the cycle all over again.

The above serves to illustrate a very important feature underlying typical AMS systems: usually, these systems involve *tight integration* between the constituent analog and digital components. Indeed, the ADC's performance (*i.e.*, its speed, power consumption, *etc.*) is frequently limited by this interplay of tight interactions. Therefore, any non-idealities in the behaviour of either the digital components or the analog components have to be carefully accounted for very carefully – because even small non-idealities can have a profound impact on performance when they propagate through the dynamic interplay between the components.

In the ADC context, this is a challenging instance of the *AMS modelling* problem, since we would like to develop models for all components that carefully account for their various non-idealities. For example, the "track and hold" unit may feature setup times, hold times, delays in tracking/holding the input waveform, *etc.*. Similarly, the DAC may feature its own delays, non-uniform discretization characteristics, *etc.*. Finally, the comparator may

exhibit small DC offsets, bandwidth limitations, input-dependent delays, *etc.*. And the digital components like the flip-flops and the logic gates in the system may exhibit their own non-ideal effects, since no digital component behaves as a perfect switch.

The challenge is to not only account for all these individual per-component non-ideal effects, but to also determine how the performance of the overall system will be impacted by these non-idealities. This is an instance of the *AMS verification* problem: given less-than-perfect analog and digital components, is it still possible to guarantee correct functionality and the required level of overall performance from the ADC as a whole?

Such questions can often be answered only by carrying out detailed (*e.g.*, SPICE-level) simulations – an instance of the *high-speed AMS simulation* problem. Furthermore, if either the simulation or the verification tool discovers a bug, one is then confronted with a non-trivial instance of the *AMS debugging* problem.

The example above shows the pitfalls and difficulties that lie along the path to bug-free AMS design – even for a relatively easy to describe AMS system such as a SAR-ADC. For higher-level systems such as SERDES systems and/or high-speed communication sub-systems, each of the above AMS problem "instances" will arguably be much harder problems to solve in their own right. This is why new approaches (such as the one proposed in this dissertation) are needed for bug-free AMS design.

1.3.2 SERDES systems and PLLs

The term SERDES stands for serializer/deserializer. Simply stated, a *serializer* accepts as input a number of *parallel* bit streams (bits travelling on different lines/wires/channels) and produces as output a single *serial* bit stream (a single line/wire/channel) that contains enough information to reconstruct the individual parallel bit streams. A *deserializer* does this reconstruction, so a serializer and a deserializer always occur in pairs (hence the term SERDES). This operation is illustrated in Fig. 1.7.

Usually, the simplest way to serialize a set of parallel bit streams is to use a high data rate for the output (the serial bit stream), and to have the output multiplex each of the input bit streams in a round robin fashion. That is, if we have 4 parallel 1 GHz bit streams at the input, the simplest serialization of these bits would produce a single 4 GHz bit stream, where a multiplexer would cycle through each of the individual parallel bit streams such that every fourth bit is drawn from the same parallel bitstream.

However, more complicated versions of the SERDES also exist, and are usually necessary to ensure performance and reliability. For example, it is possible to have a SERDES also implement some form of error control coding: an 8b/10b SERDES is an excellent example of this. By ensuring that only a small fraction of possible bit sequences are legal, 8b/10b SERDES systems make it very easy for the deserializer to detect and correct errors in the received serial bit stream.

Also, some SERDES systems interleave a clock signal (a 0 bit followed by a 1 bit) at the start of every multiplexer cycle: these are referred to as embedded-clock SERDES systems. By including a fixed bit-pattern at regular intervals, these SERDES systems typically ensure

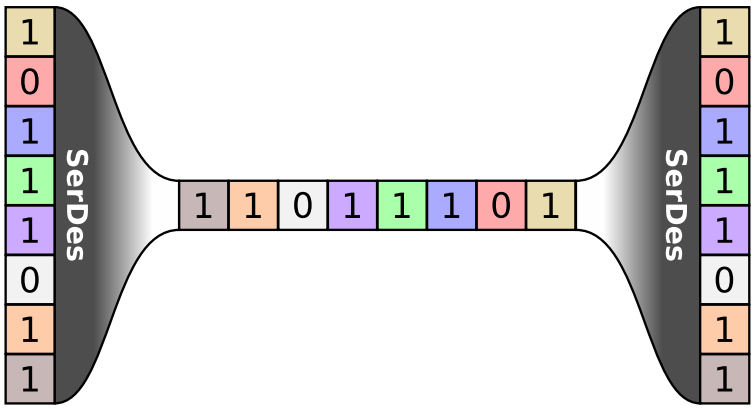


Figure 1.7: The operation of a SERDES system. The left side represents the serialization operation, where a number of parallel bit streams are converted into a single serial bit stream. This serial bit stream is then transmitted across a channel, and a deserialization operation is carried out at the other end of the channel (the right side of the figure) to recover the original parallel bit streams. Figure credits: Wikipedia.

quick recovery and return to normal operation in the event that the deserializer "slips" a cycle or two.

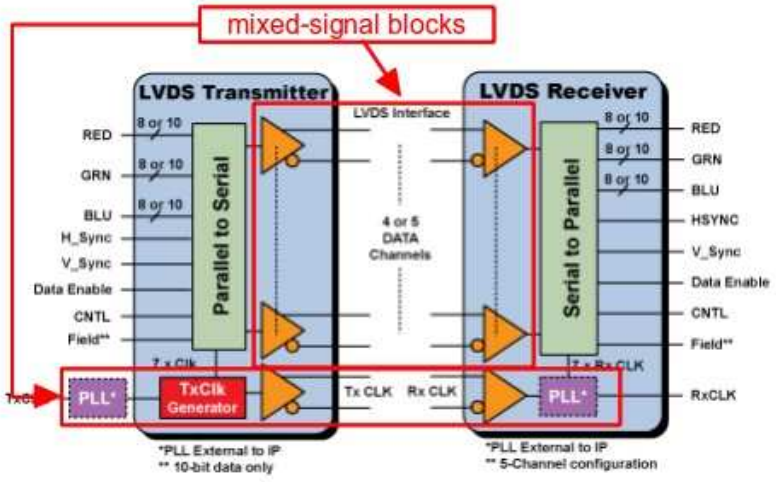


Figure 1.8: A high-level block diagram of a SERDES system.

A key challenge that is frequently encountered in SERDES systems is *clock synchronization* between the serializer and the deserializer. The deserializer needs to use a clock that is as close as possible to the clock used by the serializer: otherwise, there is a high probability that the bit streams produced by the deserializer would be "mangled". Even a small amount of systematic or deterministic clock jitter between the serializer and the deserializer

is guaranteed to eventually cause bit mangling. The apparatus is also very sensitive to random jitter and phase noise, which can quickly result in bit mangling. To overcome these non-idealities, SERDES systems often employ a variety of clock and data recovery (CDR) techniques designed to continuously monitor the deserializer clock and keep it synchronized with the serializer clock. This often involves the use of Phase Locked Loops (PLLs), which are amongst the most complicated AMS systems to model, analyze, simulate, and/or verify. A block diagram for such a typical real-world SERDES is shown in Fig. 1.8.

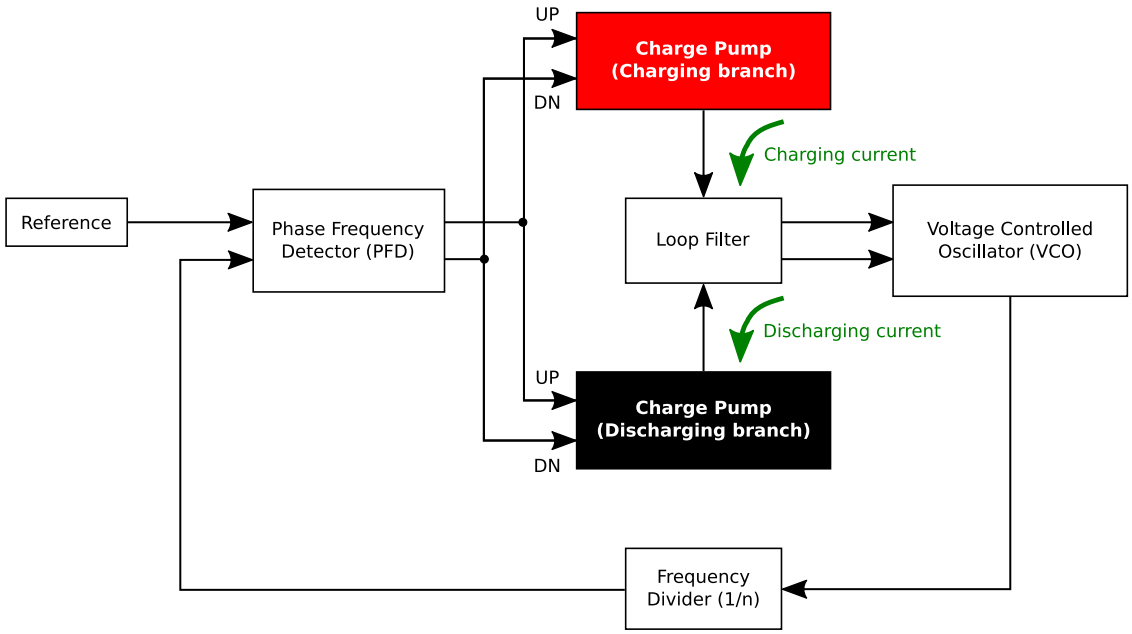


Figure 1.9: A block diagram for a simple, charge-pump based, PLL.

A block diagram for a simple charge-pump based PLL is shown in Fig. 1.9. This PLL takes as input a reference signal (such as a reference clock), and uses a feedback loop to produce an output signal (the input to the frequency divider) whose frequency and phase are related to those of the reference signal by integer multiples. For more details about how this PLL works, we refer the reader to the excellent book by Stensby [13].

As Fig. 1.9 shows, a PLL often consists of several AMS components and sub-systems. Indeed, in many modern designs, components such as the phase frequency detector, the voltage controlled oscillator, the loop filter, *etc.*, are all implemented almost entirely in digital form. However, the charge pump is often still implemented mostly as an analog circuit, and the feedback loop again leads to tight integration between the analog and the digital components (similar to the interplay in the SAR-ADC discussed above).

The tight integration above often makes formal verification of PLLs (and systems that contain PLLs, such as SERDES systems) especially challenging. For example, the PLL above is designed to produce an output signal that is supposed to "lock" to the reference signal. The question of whether this lock will occur or not is itself an extremely challenging instance

of AMS verification [14], often requiring weeks of SPICE-level simulation if a verification tool is unable to produce the requisite guarantees.

Thus, many challenging instances of AMS modelling, simulation, verification, and debugging are to be found in the domain of SERDES systems and PLLs as well.

1.3.3 A modern high-speed communication sub-system

It is no secret that high-speed communication systems play a vital rôle in the world today.

For example, in designing microprocessors at 14nm node and below, the performance of I/O (*i.e.*, the communication circuitry between the processor and memory) is critical. In many performance critical applications (*e.g.*, CAD software, audio editing, 4K video editing, gaming, virtual reality, *etc.*), large volumes of data typically need to be kept in memory and continuously transferred between the CPU cores and memory. In such applications, therefore, the performance of CPU/DRAM I/O is often the key bottleneck.

Another area where communication sub-systems play a key rôle is in wireless, *e.g.*, the smartphone and tablet revolution. There is a strong demand for ever faster and ever more power-efficient communication modules in these devices, which has led to a proliferation of wireless technologies lately: 2G, 3G, 4G LTE, *etc.*. Alongside this, there is also a parallel improvement in Wifi communication technologies: 802.11 b, g, n, ac, *etc.*.

Furthermore, the underlying communication sub-systems and infrastructure play a key rôle in data centers around the world. High speed communication equipment, including switches and routers, form the backbone of a modern data center, and several major companies and government agencies are continuously building larger and larger data centers all over the world.

Furthermore, there is a major drive towards improving the communication sub-systems underlying high performance computing systems. In virtually every parallel computing environment or cluster, communication is at least as important as, if not more important than, computation. So this is another area where high-speed communication systems are critical.

Also of note are the high-frequency trading platforms that "live and die" by the speed of their communication systems. For instance, consider the Hibernian Express, a project in progress that is building a high-speed communication network of fiber optic cables linking the New York and London stock exchanges. In this context, a mere 6 milli-second improvement in round-trip communication time was apparently sufficient to justify an expenditure of 300 million USD.

The architecture and design of such high-speed communication sub-systems are fertile grounds for carrying out AMS design research and for finding challenging instances of AMS modelling, analysis, simulation, verification, and debugging problems.

Therefore, let us devote some time to understanding, in broad terms, the architecture of a modern high-speed communication sub-system (Fig. 1.10). Perhaps not surprisingly, any communication system begins with a transmitter and a receiver, as highlighted in the figure. Also not surprisingly, there exists a channel that relays information from the transmitter to the receiver. For example, in a CPU/DRAM I/O link, this channel is a few millimetres long,

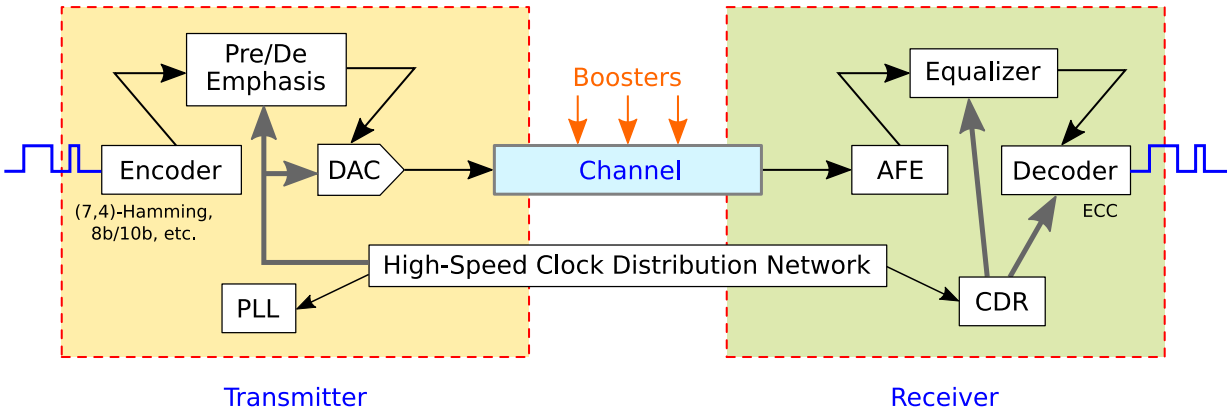


Figure 1.10: A schematic for a modern high-speed communication sub-system.

whereas in the Hibernian express, the channel is a transatlantic cable that's about 3700 miles (6000 kilometres) long.

Inside the transmitter, we have several components. For example, as the figure shows, the transmitter often uses a coding strategy to encode the bits being transmitted, so as to improve the reliability of the communication. In this case, the receiver needs to contain some decoding hardware that may implement, for example, an error detection and/or correction strategy.

Also, as Fig. 1.10 shows, the transmitter may use sophisticated techniques like pre-emphasis to improve bandwidth, which adds a few extra components on the transmit side. Also, the receiver may include some equalization circuitry at its end so as to compensate for the distortion and dispersion caused by the channel.

In this system as well, the issues of jitter, clock and data recovery, phase noise and how to manage and compensate for it, *etc.*, are prominent. This in turn ensures that there is at least one PLL in the system above, and perhaps some circuitry for clock and data recovery. These are typically included in the high-speed clock distribution network block shown in Fig. 1.10.

Furthermore, the receiver in such systems usually includes an analog front end, which typically contains one or more low noise amplifiers that magnify the received signal before it is processed further. Finally, in some communication systems, it is also necessary to install boosters/amplifiers along the channel in order to reduce signal attenuation, as Fig. 1.10 shows.

As in the other systems discussed above, a modern high-speed communication sub-system also features a high degree of interaction and tight integration between its analog components and its digital components. Indeed, it is not a stretch to say that the problem of verifying that these components all work together in such a way that the overall system satisfies a set of stringent specifications is the holy grail of the entire field of signal integrity (SI).

From the above discussion, it follows that designing a modern high-speed communication sub-system is a large and highly non-trivial AMS design problem. Indeed, in light of the importance of this problem, this dissertation devotes a substantial portion of Chapter 7

towards a new approach for carrying out a specific kind of AMS verification (namely, eye diagram analysis) for high-speed communication sub-systems.

1.4 Our approach to the AMS design problem, and an overview of the rest of this dissertation

Having described the problem of bug free AMS design, its importance and scope, and the key features that make the problem an especially challenging one to solve, this section now describes the organization of the rest of this dissertation, which presents our approach to attacking the problem.

Before describing our approach to the problem, we provide an overview of previously existing approaches. These fall under the umbrella of "hybrid system" research, and an overview of these approaches is given in Chapter 2. The central idea behind "hybrid system" approaches is to describe AMS systems using a mixture of continuous models (such as differential equations) and purely Boolean models (such as FSMs), and to develop new algorithms and techniques for analyzing and verifying these kinds of "hybrid" systems. Unfortunately, the inherent complexity of these "hybrid" systems tends to make verification problems intractable. In practice, this often necessitates accuracy compromises with regard to AMS modelling. Chapter 2 includes a detailed discussion of such tradeoffs.

Following this, in Chapter 3, we present our approach to the AMS design problem. We call our approach "Booleanization". Briefly, the core idea behind our approach is to "Booleanize" AMS systems, *i.e.*, to approximate the behaviour of AMS systems (in particular, the analog and mixed-signal components that make up AMS systems) using purely Boolean models such as FSMs. This involves developing a suite of approximation techniques capable of accepting analog formulations (*e.g.*, systems of differential-algebraic equations, SPICE netlists, *etc.*) as input and producing purely Boolean models (*e.g.*, FSMs, truth tables, *etc.*) as output. In Chapter 3, we delve into the fundamental ideas and key concepts behind this approach, and argue why it can be effective for solving AMS design problems in many situations where traditional hybrid system approaches are unsuitable.

The next 3 chapters of this dissertation (Chapters 4 through 6) are devoted to developing a suite of automated tools, techniques, heuristics, and algorithms for Booleanizing a variety of AMS systems. The first of these, Chapter 4, presents DAE2FSM, an automated technique for Booleanizing a variety of "digitalish" systems: these are systems that are intended to behave in a purely digital manner, but often end up exhibiting significant performance-limiting and error-inducing analog traits and characteristics when actually implemented in Silicon. The next chapter, Chapter 5, introduces methods for Booleanizing a large class of commonly used analog systems, namely, Linear Time Invariant (LTI) analog systems. These include I/O links, communication channels, interconnect networks, power grid networks, filters, equalizers, *etc.*, and Chapter 5 shows how all these systems can be Booleanized. Finally, Chapter 6 discusses the Booleanization of genuinely non-linear AMS systems. The

vast majority of AMS systems tend to exhibit pronounced non-linearities (in at least a few operating regions). For example, PLLs, charge pumps, certain kinds of equalizers, encoders and decoders, delay lines, ADCs and DACs, *etc.*, are all fundamentally non-linear. Therefore, for the Booleanization approach to even be applicable to such systems, we need the ability to automate the Booleanization of genuinely non-linear systems; this is the focus of Chapter 6.

Having presented our approach (based on Booleanization) to the AMS design problem (in Chapter 3), and having presented a variety of algorithms for carrying out the Booleanization of AMS designs (in Chapters 4 through 6), we devote the next chapter (Chapter 7) to presenting some concrete uses of the Boolean models produced by such methods, in the context of the AMS design sub-problems discussed above (modelling, analysis, high-speed simulation, verification, *etc.*).

Finally, in Chapter 8, we present a summary of the dissertation, highlight our conclusions, and discuss directions for future work.

Chapter 2

Previous Work

In the previous chapter, we described the problem of bug-free (*i.e.*, effectively verified and validated) AMS design and presented some motivating examples to illustrate the importance of this problem in the context of modern chip design. We also made a mention of "Booleanization", which is the approach presented in this dissertation to address the bug-free AMS design problem.

Before going on to discuss our approach (Booleanization) in detail, we would like to acquaint the readers with the current state-of-the-art in the field of bug-free AMS design. Therefore, this chapter provides an overview of previous approaches to the problem (*i.e.*, approaches and techniques that existed prior to the work presented in this dissertation). As mentioned in Chapter 1, these previous approaches typically fall under the umbrella of "hybrid system" methods for AMS modelling and verification.

2.1 "Hybrid system" approaches to AMS modelling and verification

As described in Chapter 1, AMS systems typically contain both analog and digital components. The behaviour of the analog components is usually described using differential-algebraic equations (DAEs) in the continuous domain (*e.g.*, this is how SPICE represents devices in a circuit), whereas the behaviour of the digital components is usually described using purely Boolean abstractions, such as truth tables and Finite State Machines (FSMs) in the discrete domain (*e.g.*, this is how most digital CAD tools, such as the formal verification engine ABC [6], represent the underlying design).

Also, as described in Chapter 1, the algorithms and analysis techniques that are usually used for continuous systems tend to be starkly different from (and incompatible with) those used for discrete systems. Indeed, this incompatibility between the mathematics of continuous systems and that of discrete systems is an important reason why the modelling and analysis of AMS designs is such a challenging problem: there are not many algorithms or analysis techniques that have the capability to handle both continuous and discrete mathe-

matical formulations at once. This is the goal behind "hybrid system" research – to develop algorithms and analysis techniques for systems whose components are modelled using a combination of discrete mathematics and continuous mathematics.

2.2 Illustrated example: what is a "hybrid system"?

In simple terms, a "hybrid system" is a mathematical model that features both continuous and discrete state, input, and output variables.

As we know, typical real-world AMS systems typically contain both analog and digital components. The analog components feature continuously varying voltages and currents, whereas the digital components feature purely Boolean signals (with a 1 usually interpreted as the supply rail voltage of V_{DD} Volts and a 0 usually interpreted as the ground voltage, often set to 0 Volts). This translates well to the world of "hybrid systems": the continuously varying signals in the analog components of an AMS system can be modelled as continuous variables, while the discrete signals in the digital components can be modelled as discrete/Boolean variables in a hybrid system formulation.

The equations governing the continuous and discrete variables above are usually represented in the form of a *hybrid automaton*. A hybrid automaton is a mathematical abstraction that contains a finite number of *places*. Each place is associated with a differential-algebraic equation (DAE) system. This reflects the behaviour of the continuous variables when the automaton is in a particular place. Each place is also associated with a set of *guards*: these are conditions or constraints that trigger transitions between places. When such a transition is triggered and the automaton's place changes, its continuous variables begin evolving according to a new DAE (specified by the new place reached by the automaton). This process continues indefinitely.

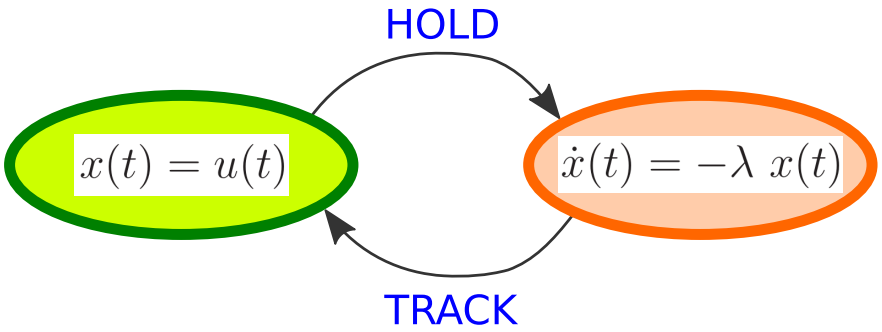


Figure 2.1: A hybrid automatom model for a "track and hold" unit featuring gradual memory loss/degradation.

The easiest way to understand the concept of hybrid automata is through an example. Fig. 2.1 depicts a hybrid automaton for a "track and hold" unit, such as the one depicted in Fig. 1.6 of Chapter 1 (Section 1.3.1). As mentioned in Section 1.3.1, an ideal "track and

hold" unit would take as input two signals: (a) an analog voltage signal (denoted $u(t)$ in the automaton above), and (b) a 1-bit digital signal (let us call this A) that specifies whether the unit should "track" $u(t)$ (if A is high) or sample $u(t)$ and "hold" the sample indefinitely (if A is low). Given these inputs, an ideal "track and hold" unit would produce as output another analog signal (denoted $x(t)$ in the automaton above), which either changes continuously in tandem with $u(t)$ (when A is high), or remains a constant value (when A is low) reflecting the sample that was recorded at the most recent moment when the input A transitioned from high to low. The hybrid automaton of Fig. 2.1 models this behaviour, with the added feature that it also models a slow loss of memory (or degradation) of the held sample over a period of time (which is a non-ideal effect that most real-world "track and hold" units exhibit in practice). This hybrid automaton contains 2 places (the green place shown on the left, and the orange place shown on the right). The green place reflects the tracking mode of operation: the DAE at this place is the simple equation $x(t) = u(t)$, indicating that the output signal $x(t)$ tracks the input waveform $u(t)$. The orange place (at the right), on the other hand, indicates the hold mode of operation featuring gradual degradation of the held value: the DAE at this place is the following equation:

$$\frac{d}{dt}\, x(t) = -\lambda\, x(t), \tag{2.1}$$

where λ is a (non-negative) constant that controls how gradually the unit's sampled value degrades over time. These DAEs are also indicated within each place in the figure. The guard conditions that specify when to transition between places are quite simple: the system transitions from the holding place to the tracking place (and vice-versa) whenever the 1-bit control input (A) transitions in the appropriate direction (from 0 to 1 and vice-versa).

Thus, the places in a hybrid automaton are somewhat akin to states in an FSM, whereas the equations in each place are akin to SPICE-like DAEs. The guard conditions act like controllers that "sense" the underlying state variables and input variables, and "actuate" abrupt/discrete changes by transitioning the system from one place to another. In this way, the hybrid automaton abstraction allows one to model systems that feature a combination of both continuous and discrete state, input, and output variables.

2.3 A brief overview of previous approaches to bug-free AMS design

As mentioned above, a variety of approaches involving hybrid systems and hybrid automata have been developed for modelling, analyzing, and verifying AMS systems. Indeed, a substantial body of work exists on AMS analysis and modelling using hybrid system approaches, and much has been published on the topic by research groups at various universities, as well as by AMS design/CAD groups in commercial chip-design houses and corporations. As a result, this literature is too vast to cover in full detail here; however, we will provide a

brief overview highlighting the key tools and techniques developed in this area, and how Booleanization (the approach proposed in this dissertation) can nicely complement them.

This section is organized as follows: we begin with a "breadth-first" general overview of existing hybrid system approaches, without delving into too much detail on any specific method. The goal behind this exercise is to identify common characteristics (as well as limitations) underlying hybrid system methods in general, while covering a variety of approaches. After this, we shift gears and go "depth-first": we pick out a moderately successful approach from the literature (a paper that uses continuization techniques to verify PLLs) and study the approach in some detail. This allows us to substantiate our general assertions and observations about hybrid system methods, by enabling us to appropriately transfer emphasis from the generic to the specific.

2.3.1 A general overview

The AMS modelling, analysis, and verification problems described above have received considerable attention from both Boolean and hybrid system researchers over the past several years.

At the outset, we would like to make an observation about the kinds of DAEs that are typically featured in hybrid automata. Because hybrid automata are typically intended to be formally verified, and because this verification is inherently a very difficult problem (one that is often intractable, and at times even undecidable), most hybrid automata are constructed *after* making a number of simplifying assumptions (resulting in so called "behavioural models" of AMS components). For example, in the hybrid automaton of Fig. 2.1, we made the simplifying assumption that the "track and hold" unit does the tracking part perfectly, regardless of how quickly the input waveform changes or how much it swings; this is unrealistic because most real-world "track and hold" units feature bandwidth limitations and operate properly only when the input stays within a relatively narrow range. In practical terms, this means that the DAEs associated with most hybrid automaton places tend to be rather simplistic, often featuring no more than a handful of continuous variables. Often, these DAEs can be linearized approximations to the underlying system. Thus, hybrid automaton DAEs for typical AMS components are not as detailed, or as accurate, as the corresponding SPICE-level DAEs for the same component. Such *a priori* modelling simplification is often carried out manually – so the end-user would be well-advised to closely examine the hybrid automaton DAEs to ensure that they are accurate enough for the AMS modelling/verification problem at hand.

To take a more concrete example from the literature, consider the work by Ghosh and Vemuri [15], which was one of the first contributions towards AMS verification. Ghosh and Vemuri applied the PVS proof checking tool to analog circuit models. However, these models, like the hybrid automaton DAEs above, were somewhat simplistic. For instance, they used idealized OpAmps, transistors with constant transconductance, *etc.*. While this was an important step towards AMS verification, the range of applications of this method was limited because of these modelling simplifications. This issue could not be addressed by making the

DAEs more detailed because this led to computational challenges arising from the need to formally verify arithmetic over real numbers. This is yet another example of the tradeoff between modelling accuracy and amenability to analysis, as described in Section 1.2.5.1.

Soon after the work by Ghosh and Vemuri, the pace of AMS design and verification research quickened. A key thrust of this research was to develop new techniques for abstracting analog dynamics into highly simplified behavioural models, driven by the inherent scalability limitations of verifying continuous systems. A key feature of AMS modelling/verification research in this period is that these behavioural models were often carefully tailored to specific and narrow application domains/circuit classes. For example, Hanna and others [16,17] developed new approaches specifically to verify digital circuits suffering from analog non-idealities.

In parallel, there was (and still is) another branch of AMS verification research seeking to model continuous dynamics in a more general fashion, without being as system/application specific: these methods typically work by partitioning the continuous analog state space of voltages and currents into discrete *domains*, and the idea is to encode transitions between these domains using Boolean data structures like FSMs, BDDs[1], *etc.*. For example, the work by Kurshan and Macmillan [18], Hedrich *et. al.* [19,20], *etc.*, fall into this category.

The formal verification and model checking of such relatively general-purpose abstractions also received attention from the community. One solution was to use off-the-shelf Boolean verification techniques with small modifications (*e.g.*, as exemplified by Kurshan in [18]). Another approach was to augment existing CTL[2] model-checking tools with a variety of extra AMS-relevant features (as espoused by Hedrich and others [19,20]). While the former approach was simpler in principle, the latter approach allowed for greater flexibility in model and property simplification, at the cost of significant code rewrite/development effort.

However, in spite of researchers' best efforts along both these lines, their techniques were scalable only to small designs (*e.g.*, a single gate [18], or a small tunnel diode [19]). Also, owing to the ever-present tradeoffs between modelling accuracy and amenability to analysis (Section 1.2.5.1), these techniques tended to be limited in their power to model real-world analog phenomena that were of interest to AMS designers. In addition, there were no sufficiently automated methods for deriving even such simplified abstractions from general SPICE-level netlists, since the model-generation phase of these approaches often relied heavily on manually generated simplifications.

With a view to addressing some of these limitations and scaling up AMS verification techniques to real-world designs, several new *modelling frameworks* were introduced. These modelling frameworks often provided well-defined *mathematical abstractions* for representing and reasoning about hybrid systems and automata. In addition, some of them provided *languages* for expressing AMS-design relevant properties and predicates, in many cases with associated verification methodologies and reachability analysis algorithms. Today, these frameworks fall under the umbrella of "hybrid system verification", a topic that has been

[1]Binary Decision Diagrams.
[2]Computation Tree Logic.

extensively studied in the literature, and mathematically formalised by notable researchers such as like Alur, Nerode, and Henzinger [21–24].

The modelling/formalisation efforts above were critical because many reachability questions on general hybrid systems and automata are, in fact, undecidable (provably computationally equivalent to the halting problem [12]). Therefore, for AMS verification to even be possible (let alone practical), it is necessary to either restrict one's domain to classes of hybrid systems that are known to be decidable, or accept that the verification flow would be largely incomplete; the above formalisation efforts helped develop a strong *theory of hybrid systems* that enabled researchers to ask more meaningful questions about decidability and time/space complexity, which in turn spurred new advances in reachability analysis/model checking of AMS systems.

For example, Al-Sammane *et. al.* used recurrence relations and difference equations to simplify analog blocks [25], which were later verified using interval arithmetic and Taylor approximations [26]. This methodology was successfully used to check the stability of a third order $\Delta\Sigma$ modulator (modelled behaviourally). Also of significance is the d/dt tool, developed exclusively for model checking continuous linear systems [27], although limited to rather small system sizes.

Other novel hybrid system modelling frameworks (that either restrict themselves to decidable classes of hybrid systems, or offer highly incomplete verification flows) for AMS include: guarded state machines as applied to flash memories, verified using the ACL2 theorem prover [28], linear hybrid automata verified using tools like HyTech [29] and Phaver [30], polyhedral invariant hybrid automata, verified using flowpipes and the theory of quotient transition systems, by tools like Checkmate [31], labelled hybrid petri-nets (or LHPNs), verified using difference bound matrices as part of the LEMA toolkit developed by Chris Myers *et. al.* [32–35], and many more that are too numerous to mention here. In addition, several new algorithmic refinements have improved the accuracy and efficiency of hybrid system reachability analysis. These include zonotopes [36], hybrid restriction diagrams [37], and other over-approximation techniques (*e.g.*, using support functions [38], continuization methods [14], *etc.*).

2.3.2 A specific example: PLL verification via continuization

To take another example of an AMS verification problem that is well-studied in the literature, consider the problem of verifying that a PLL locks to its reference under all appropriate operating conditions. As described in Section 1.3.2, this is a very important and desirable property, since a PLL that does not lock severely impacts the performance and bit error rate (BER) of any system that relies on it for clock and data recovery (*e.g.*, high-speed communication sub-systems, SERDES systems, *etc.*). Also, this instance of AMS verification is a particularly challenging problem because PLLs are amongst the most complex non-linear dynamical systems encountered in AMS design that are difficult to analyze, simulate, and verify.

For example, a state of the art hybrid system based method for Phase Locked Loop (PLL) verification is presented in [14]. This work attempts to formally verify that a given charge-pump based PLL will eventually lock to a desired limit cycle.

Fig. 2.2 depicts the PLL schematic used in the work above. To the authors' credit, this is indeed a reasonable high-level representation of a PLL.

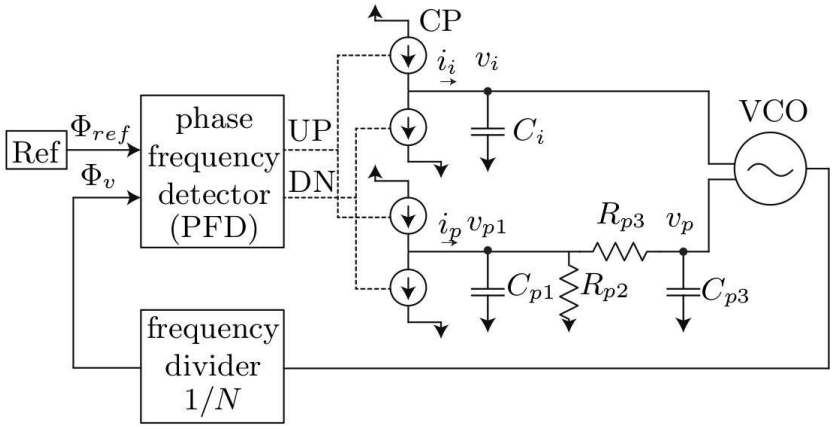


Figure 2.2: Schematic of a charge pump PLL as used in [14]. Figure credits: M. Althoff, *et. al.* [14].

However, for verification to be feasible, the authors are then forced to drastically oversimplify the complex dynamics exhibited by the system of Fig. 2.2. For instance, real-world PLLs such as the one depicted above typically contain about $500,000$ (or more) transistors. This makes for a highly non-linear system with extremely complicated dynamics. But the authors of [14] reduce this $500,000$ size system to a set of *linear* DAEs of size 5. Thus, the model that actually ends up getting verified is so far removed from transistor-level reality that it is impossible for an AMS designer to have confidence in the "guarantees" produced by the verification tool. Furthermore, there is no automated procedure to derive the simplified model equations from a more detailed transistor-level or SPICE-level model: the process appears to be largely manual.

To their credit, the authors of [14] do recognize and acknowledge some of the shortcomings above. However, there is no known technique available to address these issues. As mentioned above, hybrid system models are often forced to be extremely small and simplistic because currently known hybrid system verification tools and techniques are incapable of handling more complicated models. Boolean verification techniques, by contrast, have evolved to a point where they can handle much larger and more complicated models, which is why we believe that Booleanization offers a more promising route to attack the AMS verification problem.

Furthermore, consider the hybrid automaton used by the authors of [14] to control the charging/discharging dynamics of the charge pump in the PLL (Fig. 2.3). This is also a very

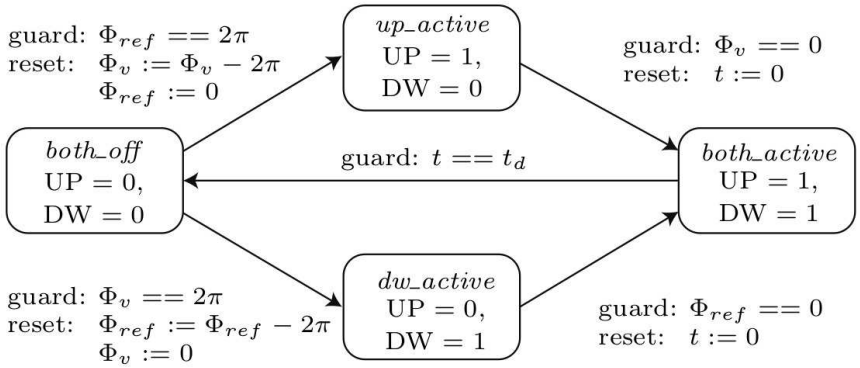


Figure 2.3: Hybrid automaton used in [14] to control charge pump switching. Figure credits: M. Althoff, *et. al.* [14].

simplistic model that is roughly comparable to "back of the envelope" calculations, *i.e.*, a model that may not be well-suited for formal verification. In reality, the signals `UP` and `DOWN` would exhibit timing mismatches, non-ideal waveform shapes, *etc.*, and they would depend on the underlying voltages and currents in the circuit in a complicated way that bears only a passing resemblance to the neat phase equations used in [14], as described below.

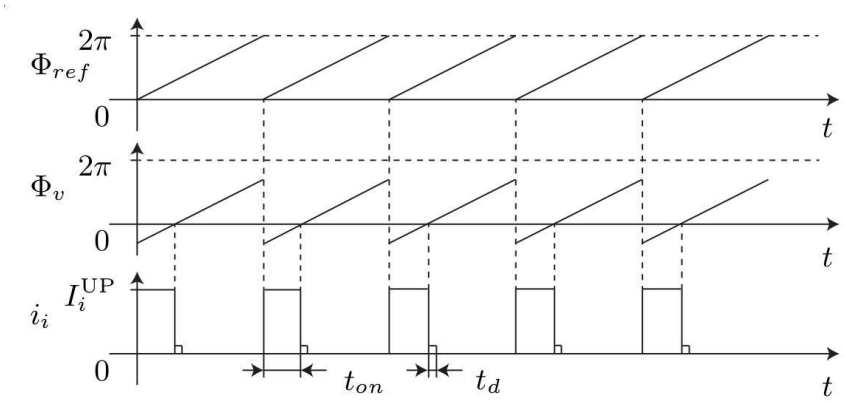


Figure 2.4: Typical simulation traces exhibited by the hybrid automaton of Fig. 2.3. Figure credits: M. Althoff, *et. al.* [14].

Indeed, the equations used by the authors in [14] do not model the underlying system in terms of fundamental quantities like voltages and currents. Instead, the equations directly quantify changes in the *phase* of the PLL output with respect to the phase of the reference signal. Phase is a highly derived quantity whose relationship to the underlying voltages and currents is very non-linear, complicated, and difficult to model. However, the PLL model that is ultimately verified in [14] ignores all these real-world complexities, and naïvely

assumes that the PLL's phase always evolves with constant slope and that it perfectly aligns with the `UP` and `DOWN` signals received by the charge pump (as seen from Fig. 2.4) with no timing delay, non-linearity, charge pump saturation, current mismatches/offsets, or any of the myriad other real-world effects that have crucial impact on PLL locking.

However, in spite of these shortcomings, [14] in fact represents one of the best attempts to date to apply hybrid system techniques to PLL verification.

2.4 Limitations of previous approaches

As touched upon above, we believe that previous approaches to AMS modelling and verification suffer from 3 main inter-related drawbacks, (1) limited AMS modelling accuracy, (2) lack of automated modelling tools and techniques, and (3) scalability limitations when it comes to verification.

2.4.1 Limited AMS modelling accuracy

As we mentioned above, most existing AMS verification approaches only work with simplified behavioural models for the underlying AMS components; these models often consist of only one or two continuous signals, and the dynamics exhibited by these models are also only simplified approximations to the real design.

For example, in most hybrid automata, non-linear dynamics are often ignored and replaced by crude linearized approximations. Even when non-linearity is considered, it is usually modelled by piecewise constant-slope approximations, *etc.*. Transistors, ADCs, DACs, *etc.*, are usually assumed to be ideal. Important performance-limiting analog details such as waveform shapes, inter-symbol interference, distortion and dispersion, DC offsets, delays and bandwidth limitations, *etc.*, are typically ignored.

Thus, the device/component models used for AMS verification are often very different (and less detailed and of lower accuracy) compared to the SPICE-level models that are typically used for AMS simulation. We believe that this reduced accuracy is an important limitation, and we also believe that the techniques proposed in this dissertation can overcome some of these accuracy limitations. Indeed, we believe that fixing the accuracy limitations of existing AMS modelling approaches is the most important contribution of ABCD.

2.4.2 Lack of automated AMS modelling tools and techniques

As we stated above, most existing tools and techniques for AMS verification make use of *a priori* modelling simplifications. Furthermore, it is also troubling to note that many of these behavioural modelling simplifications are not derived in any systematic or automated way; rather, the process of obtaining AMS macromodels for hybrid automata is one that is highly manual, and one that calls for a deep understanding and intuition about not only which aspects of an individual AMS component's behaviour will affect its own behaviour, but also

which aspects will impact the functionality, correctness, and performance of the overall design with its tightly integrated interactions between analog and digital components. We believe that this level of "black magic" is undesirable and dangerous: for example, it often leads to false positives and false negatives when manually generated AMS behavioural models that fail to capture important effects are used in a verification flow. Also, the resulting models are unlikely to capture non-ideal effects and system behaviours that the designer is either unaware of, or incorrectly considers not important enough to watch out for, which can compromise the main goal of verification.

By contrast, this disssertation proposes a suite of highly automated techniques for AMS modelling, which we believe will not only produce more accurate models, but will also help eliminate "designer bias" and untested corner cases during verification.

2.4.3 Scalability limitations for verification

Finally, we recognize that the scalability of verification is an important limitation for the current generation of AMS verification tools and methodologies. Indeed, this is an important reason why one is forced to adopt highly simplistic macromodels for AMS components in the first place.

Scalability limitations of existing techniques arise largely from the continuous variables used while modelling AMS systems. The purely Boolean/discrete variables, by contrast, do not exacerbate the scalability problem to such a great degree: indeed, purely Boolean verification engines are routinely used in the industry for reachability analysis and sequential equivalence checking of circuits with millions of gates and several hundred flip-flops. Such kinds of systems are well beyond any existing AMS verification approach.

Based on the fact that scalability issues arise mainly from the continuous variables in AMS verification problems, the approach we propose (namely, Booleanization) aims to represent as much of behaviour of the underlying AMS components as possible using purely Boolean variables. This, therefore, can help avoid the severe penalties imposed by the need to use continuous variables while modelling AMS components.

2.5 How Booleanization can complement previous approaches

From the discussions above, it is clear that much effort has been devoted to developing and fine-tuning hybrid system modelling frameworks and verification engines for AMS systems. However, it is also true that existing AMS modelling and verification methods often suffer from accuracy and scalability limitations. We believe that the tools and techniques developed in this dissertation (namely, Booleanization) can effectively address some of these limitations.

As pointed out above, one trait that is shared by almost all existing AMS verification methods is that they work with simplified behavioural models for AMS components – models and DAEs that do not bear close resemblance to SPICE. The main reason for this is

scalability. Indeed, while many hybrid frameworks have been proposed for carrying out reachability analysis of systems involving both discrete and continuous quantities, the verification problem for continuous quantities often scales much more poorly than the traditional verification problem that only involves purely Boolean/discrete quantities. This is the key factor that limits the applicability of such frameworks to small (often linearized) DAEs and simple behavioural models of AMS systems, rather than models that achieve SPICE-level accuracy.

We believe that the question of *SPICE-accurate modelling* of AMS components, which ensures that the circuit models used by verification engines actually reflect underlying analog reality while still striving to remain scalable from a verification/reachability analysis standpoint, has not received adequate scrutiny in the existing AMS modelling/verification literature. Without SPICE-accurate modelling, the predictions made by verification engines are questionable and dangerous for designers to rely on. Indeed, this is an important reason why the prevailing practice amongst AMS designers today is to carry out time-consuming SPICE simulations rather than place their trust in AMS verification tools.

To overcome such "designer skepticism", we believe that it is necessary to, (a) significantly scale up existing hybrid system techniques, so that they embrace SPICE-accurate models even for large AMS designs, and (b) develop new automated techniques for constructing such accurate models, starting from SPICE-level circuit descriptions, that are amenable to such scalable verification. Therefore, the automated generation of near SPICE-accurate models for AMS components that are simultaneously suitable for verification as well, is an important thrust of this dissertation.

The key idea that we advocate is this: to avoid scalability issues, we suggest that continuous variables (which introduce major computational challenges that are orders of magnitude more severe than purely discrete/Boolean variables) be used as sparingly as possible in the modelling of AMS components.

Thus, in our view, there is a need to develop new techniques that accurately capture the behaviour of continuous systems, while refraining from the use of continuous variables. This is the essence of Booleanization. In other words, we would like to approximate continuous systems to a high degree of accuracy while only using purely discrete/Boolean variables (even though, in theory, hybrid system approaches can represent and reason about both kinds of variables). Armed with a powerful suite of Booleanization techniques (such as the ones proposed in Chapters 4 through 6 of this dissertation), we believe that much of an AMS system's behaviour will be representable using "cheap" purely discrete/Boolean variables, which frees up the "precious" continuous variables for use only when absolutely necessary. This, in turn, may help alleviate the scalability issues inherent to existing hybrid system approaches, and may one day enable the verification of large AMS systems at or near SPICE-level accuracy.

We view this dissertation as a step in the above direction. We believe that if one were to integrate the Booleanization techniques proposed herein (Chapters 4 through 6) with existing Boolean and hybrid system frameworks for AMS verification, one would be able to significantly expand the scope of existing AMS verification techniques to handle much larger

systems than they can do so at present. This is our hope for the future of AMS verification, and is the larger context behind this dissertation.

Chapter 3

Booleanization: Our Approach to AMS Design

This chapter introduces the concept of "Booleanization" of a continuous system. Booleanization is the central idea underpinning this dissertation, and it represents our approach to the longstanding problem of bug-free AMS design, modelling, analysis, fast simulation, verification, and debugging.

Booleanization refers to the practice of approximating continuous (and mixed-signal) systems using purely Boolean models, while striving to maintain accuracy and scalability.

This chapter covers the "what" and "why" of Booleanization in detail, *i.e.*, the core concepts underlying Booleanization are explained, and the motivation for Booleanizing AMS designs, in the context of effective, bug-free AMS design, is discussed. The "how" of Booleanization (*i.e.*, the development of algorithms and techniques for carrying out the Booleanization of real-world AMS designs), on the other hand, is not addressed in this chapter; this is the subject of the next three chapters (Chapters 4 through 6).

Organization of this chapter: We begin this chapter with an explanation of what Booleanization is, and the high-level ideas behind how Booleanization typically works (Section 3.1). We then delve into finer details such as what constitutes a Boolean model (Section 3.2), the differences between Booleanization and discretization (Section 3.3), the precise relationships between AMS systems and their Booleanized versions (Section 3.4), *etc.*.

Having thus described the core concepts underlying Booleanization, we then present a concrete example (Section 3.6) of the process of Booleanization in action; this example serves to illustrate the core concepts behind Booleanization in a clearer way, thereby filling gaps in the reader's understanding of Booleanization. The example (Section 3.6) starts with an analog system (an RLGC filter described as a system of differential equations), Booleanizes it (using one of the techniques developed later in this dissertation), and compares the behaviour of the resulting Boolean model against that of the original analog system.

Motivated by the example above, we also discuss a key tradeoff involved in Booleanization, *i.e.*, the tradeoff between accuracy and size of the resulting Boolean model. This has implications on how to clock the Boolean model, as well as how finely to discretize continuous

signals while Booleanizing an AMS system.

Next, we address the question of why we would want to Booleanize AMS systems in the first place. In other words, how will Booleanization help us solve the AMS design problem? In a nutshell, Booleanization promises to bring better scalability to AMS verification (because the problematic continuous quantities have been eliminated from the system description) while simultaneously retaining a reasonably high degree of modelling accuracy. This is covered in Section 3.8.

We then continue with a discussion on the need for developing automated Booleanization techniques, and the strengths and weaknesses of automated Booleanization techniques with respect to manual ones (Section 3.9).

Finally, we this chapter by introducing ABCD, a suite of automated Booleanization techniques developed by us for several different kinds of AMS systems. These Booleanization techniques (comprising DAE2FSM, ABCD-L, and ABCD-NL) form the basis of the next few chapters.

3.1 What is Booleanization?

In a nutshell, Booleanization means taking a continuous (or mixed-signal) system and coming up with a purely Boolean approximation for it. That is, given an *analog* (or *mixed-signal*) system S_A that operates in continuous (or mixed-signal) space and time (*e.g.*, a system of differential-algebraic equations), we would like to *approximate* S_A by a new *Boolean* system S_B that operates in discrete space and time (*e.g.*, a Finite State Machine (FSM)).

We note that, both in principle and in practice, Booleanization is very different from the hybrid system approaches that we described in Chapter 2. Indeed, the premise itself is different. Hybrid system approaches work with abstractions (such as hybrid automata) that have the capability to represent both continuous and discrete quantities and variables. On the other hand, with Booleanization, our abstractions (such as FSMs) are only capable of representing Boolean/discrete quantities. There is no notion of continuous space or time in a Booleanized system.

For example, in the hybrid automaton shown in Fig. 2.1, the variables u and x are continuous quantities, whereas the input that decides whether the unit should track or hold is a Boolean quantity. If we were to Booleanize this system, we would have to figure out an alternative way to represent the continuous signals u and x that makes use of only Boolean quantities.

At first glance, this might seem like a crippling limitation. After all, if we want to model and analyze AMS systems featuring both continuous and discrete signals, it is not immediately obvious that we can model our systems of interest with sufficient accuracy while restricting ourselves only to Boolean quantities. Closer examination, however, suggests that this is indeed possible: we can discretize any continuous signal in our system and represent it as a vector of bits. The greater the number of bits used to quantize the signal, the greater the accuracy with which we can capture the signal's time-varying waveforms. In a sense,

therefore, given a desired level of accuracy, we can always achieve this accuracy, at least in principle, just by increasing the fineness of discretization while Booleanizing our system.[1] For a more detailed discussion of the tradeoffs involved between accuracy and Boolean model size, please see Section 3.7.

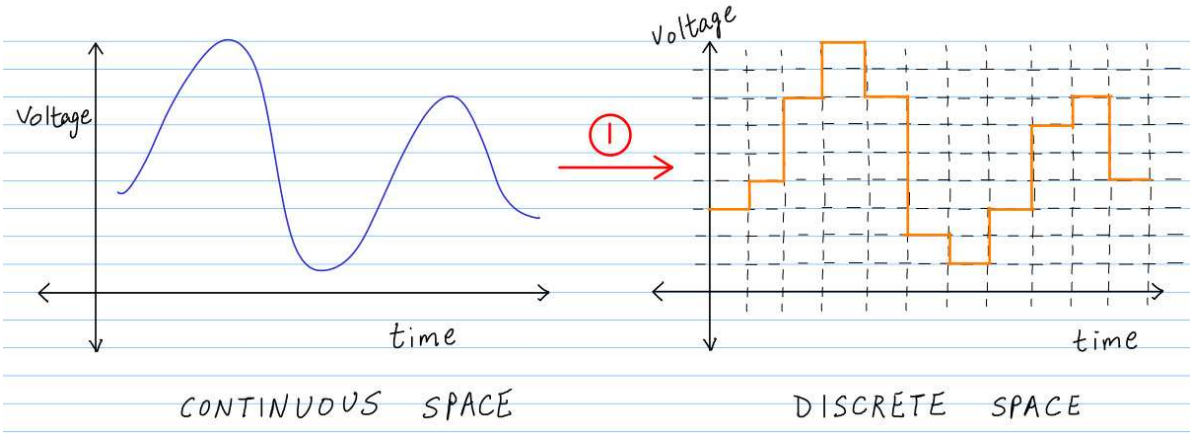


Figure 3.1: The first step in the process of Booleanization is to discretize all continuous quantities of interest in the given AMS system S_A. The blue waveform on the left represents such a continuous signal, and the orange waveform on the right is its discretized version.

Thus, the first step in the Booleanization process typically involves discretizing every continuous quantity of interest in our original AMS system (S_A), so that the Booleanized system that we construct (S_B) features only Boolean/discrete quantities. This is illustrated in Fig. 3.1. At the end of this step, we have transitioned all our variables (*e.g.*, voltages and currents in an AMS system) from a continuous/mixed-signal space to a purely Boolean/discrete space. For an example, please see Section 3.6. For more details on the discretization process, please see Section 3.3.

The next step in the Booleanization involves handling the dynamics associated with the continuous quantities in our AMS system, *i.e.*, modelling the DAEs that govern the evolution of these continuous quantities (like voltages and currents) over time. In the pure SPICE and hybrid system worlds, representing these DAEs as continuous systems of equations is already supported. However, in our discretized Boolean state space, it is not immediately clear how to model these DAEs, because we do not have access to any continuous variables.

[1] We believe that this holds true as long as the underlying system being Booleanized is reasonably well-behaved, *i.e.*, it satisfies basic existence and uniqueness properties, it features reasonably smooth waveforms, it is robustly stable and non-chaotic, *etc.*. While one can certainly construct DAEs that do not satisfy these conditions and are therefore extremely difficult to Booleanize accurately even when using a large number of bits to represent the underlying signals, fortunately, typical AMS systems encountered in practice comfortably satisfy all these requirements and are fairly easy to Booleanize with high accuracy, by using a sufficiently large number of bits to discretize circuit signals.

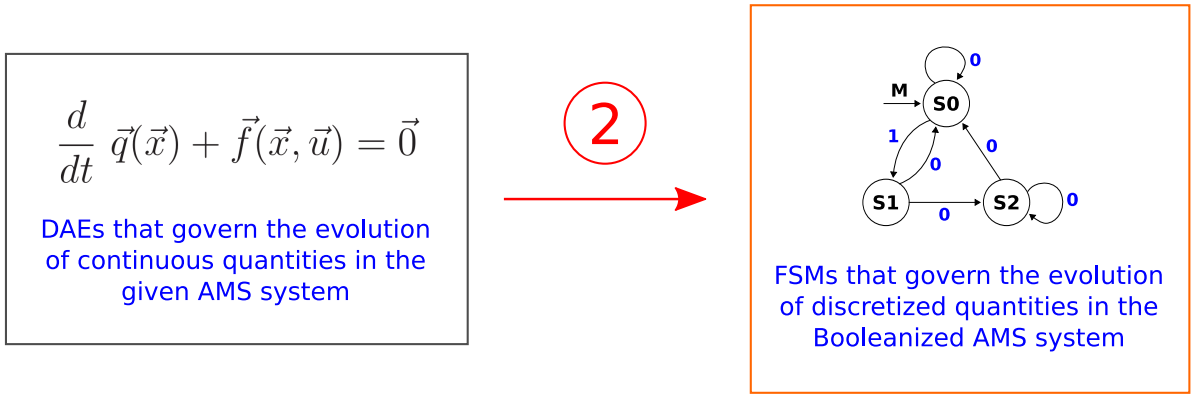


Figure 3.2: The next (second) step in the process of Booleanization is to construct a purely Boolean model (such as an FSM) that mimics the DAEs in the original AMS system (S_A), but employs only Boolean constructs, operations, and abstractions (*e.g.*, Boolean full-adders and other combinational logic, counters, registers, *etc.*) on the discretized signals formalised in the previous step.

The resolution is that we can use FSMs to approximate the DAEs instead. That is, we can "mimic" the behaviour of the DAEs in the original system S_A by carefully constructing purely Boolean FSMs that operate only on the purely Boolean signals in the discretized state space formalised in the first step above. This is illustrated in Fig. 3.2. Thus, Booleanization is not the same as discretization, but discretization is typically the first step in the process of Booleanization. For a more thorough discussion of the differences between discretization and Booleanization, please see Section 3.3.

3.2 What constitutes a Boolean model?

Since the rest of this dissertation relies heavily on the notion of Boolean models, we would first like to take a moment to describe what we mean by a Boolean model.

A Boolean model is essentially any system that operates only on Boolean quantities. For example, many purely digital circuits can be modelled as Boolean systems, using abstractions such as truth tables, FSMs, *etc.*, as explained below.

Here, it is useful to make a distinction between *combinational* Boolean systems and *sequential* Boolean systems. Combinational Boolean systems do not have a notion of state. They just map Boolean inputs to Boolean outputs. In other words, a combinational system's current outputs can always be fully determined from its current inputs. If the inputs do not change, neither will the outputs. Also, a given input (a specific vector of bits) always produces the *same* output (another vector of bits), regardless of when the input is applied to the system. That is, the system has no memory of previous inputs while it determines its current output. Fig. 3.3 shows an example of a combinational system, a full-adder circuit.

This circuit takes in 3 Boolean inputs (A, B, and C_{in}), and produces as output 2 Boolean outputs (S and C_{out}).

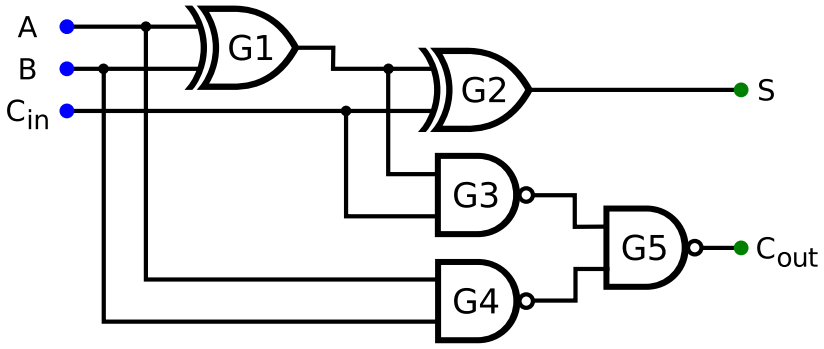


Figure 3.3: A full-adder circuit (an example of a combinational system).

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 3.1: The truth table for the full-adder circuit of Fig. 3.3.

A combinational Boolean system can be fully specified by its *truth table*, a simple table that maps all combinations of system inputs to their outputs. For example, Table 3.1 shows a truth table for the full adder circuit of Fig. 3.3. This truth table lists the outputs (S and C_{out}) produced by the full adder for each of the 8 possible combinations of the inputs (A, B, and C_{in}). Since the truth table completely specifies this system's behaviour, we say that it is a *Boolean model* for the underlying system.

However, truth tables are not the only kinds of Boolean models for combinational systems. More efficient models, such as Binary Decision Diagrams (BDDs) [39], And Inverter Graphs (AIGs), *etc.*, are typically used in practice for fully specifying combinational systems. For example, Fig. 3.4 shows a BDD that represents the full adder circuit of Fig. 3.3. This is an equally valid Boolean model because it also fully specifies the behaviour of the circuit. Also, given a BDD, one can generate the truth table from it and vice-versa.

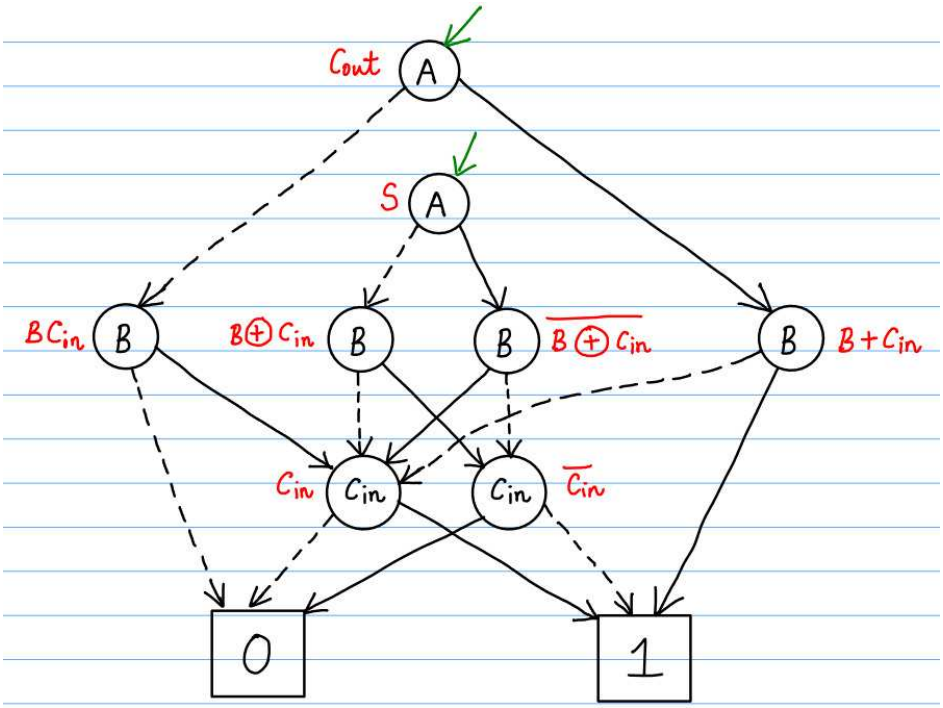


Figure 3.4: A Binary Decision Diagram (BDD) [39] for the full-adder circuit of Fig. 3.3. The green arrows indicate the output nodes for S and C_{out} of the full-adder, and the red labels indicate the Boolean function implemented by each intermediate node in the BDD.

Sequential Boolean systems, on the other hand, do have a notion of *state*. The output produced by a sequential system at a given (discrete) time can depend on not just the current input, but also the history of inputs leading up to the current input.

A common way to model sequential Boolean systems is by using Finite State Machines (FSMs). For example, Fig. 3.5 shows a simple FSM that models a sequential Boolean system. As the figure shows, the FSM has a finite number of *states* (hence the term finite state machine), and the sequential system at any given time can be in any of these states. Initially, the system is in a specially designated state called the *start state*. For example, the FSM of Fig. 3.5 has 3 states named S_0 through S_2, and its start state is S_0.

The FSM works by executing a sequence of *transitions*, as follows. As mentioned above, at each point in (discrete) time, the system is in one of the FSM states. At this state, the system obtains a discrete *input symbol*. In the case of the FSM in Fig. 3.5, the input symbol is either 0 or 1. Having obtained this input symbol, the FSM then *transitions* to a new state (called the destination state), and produces an *output symbol* in response to the input symbol. The rules that decide the destination state and the output symbol, given the current state and the input symbol, are specified as *arcs* of the FSM. For example, the FSM in Fig. 3.5 has 2 arcs leading out of every state. Each arc specifies a green input symbol and a blue output symbol. For example, consider the arc that reads 1/0 that originates at S_1 and ends

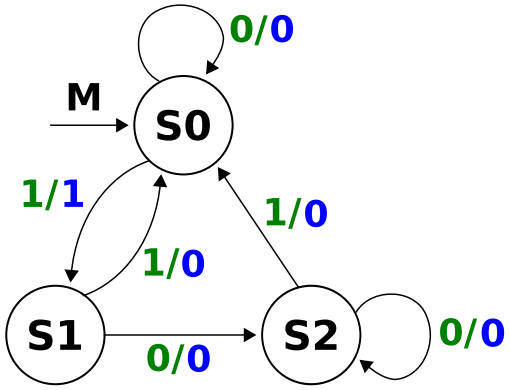


Figure 3.5: A simple FSM that models a sequential Boolean system. The FSM has 3 states named S_0 through S_2, and its start state is S_0.

at S_2. This arc indicates that if the system is at state S_1, and it encounters the input symbol 1 when it is at S_1, the system can transition to state S_0 while producing the output symbol 0. Thus, in general, the current output symbol produced by an FSM depends not only on the current input but also the current state (and the current state is itself a function of the previous inputs encountered by the FSM). This kind of FSM is referred to as a Mealy machine [12]. There is also another kind of FSM where the current output depends only on the current state, and not on the current input; this is called a Moore machine [12]. Unless otherwise stated, all FSMs in this dissertation are Mealy machines.

Current state	Input symbol	Next state	Output symbol
S_0	0	S_0	0
S_0	1	S_1	1
S_1	0	S_2	0
S_1	1	S_0	0
S_2	0	S_2	0
S_2	1	S_0	0

Table 3.2: A state transition table for the FSM in Fig. 3.5.

As with truth tables for combinational systems, one can fully specify the behaviour of sequential systems using *state transition tables*. For example, Table 3.2 shows the state transition table for the FSM in Fig. 3.5.

As with truth tables, storing the state transition relation of an FSM as a table or as a graph is in general not efficient. Therefore, a variety of specialized data structures have been developed to efficiently represent and manipulate FSMs. For example, a commonly used model for sequential systems is the so-called Boolean circuit representation (Fig. 3.6).

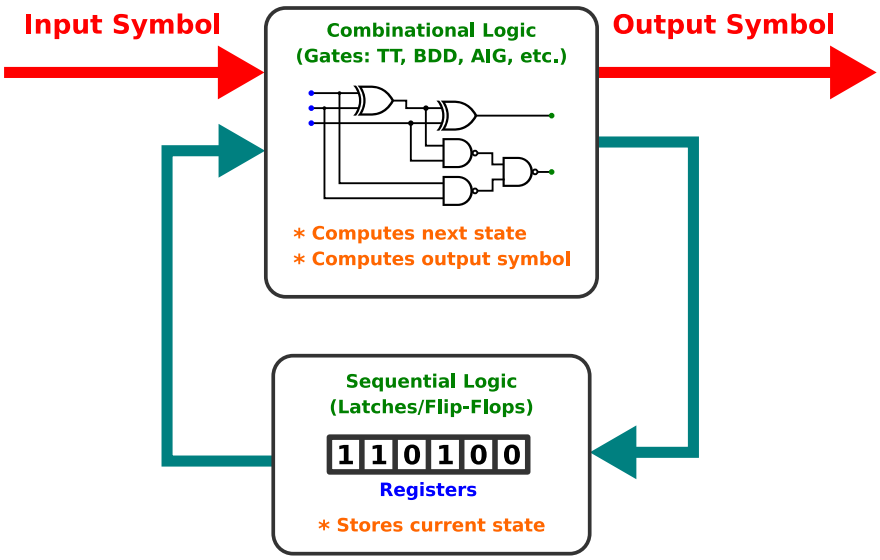


Figure 3.6: Boolean circuit representation of an FSM, consisting of both combinational logic and sequential logic.

It consists of a set of *state registers* that store the current state of the sequential system, and a *combinational logic block* that computes the next state of the system (to which the state registers are updated) and the output symbol based on the current state and the input symbol. The state registers are *clocked*, meaning that they get updated periodically at the rising/falling edge of a clock signal. A variety of specialized data structures, such as And Inverter Graphs (augmented with latches/registers) are typically used in practice to store and manipulate FSMs in the form of Boolean circuits. For example, such AIGs are the data structure of choice for the state-of-the-art verification tool ABC [6].

To summarise, in the rest of this dissertation, when we use the term "Boolean model", we are referring, in abstract mathematical terms, to either a truth table (for a combinational Boolean system) or an FSM (for a sequential Boolean system). However, this is only a convenient mathematical abstraction. In practice, we often use more efficient data structures for representing our Boolean models. For example, instead of storing a truth table, we might store a BDD or an AIG. These are equivalent to a truth table from a mathematical perspective (since they have the same expressive power as a truth table, and since BDDs, AIGs, and truth tables can all be converted from/into one another), but are much more efficient in practice. Similarly, instead of storing the state transition graph or table of an FSM, we may decide to store a Boolean circuit representation (such as an AIG augmented with latches/registers) in the interest of efficiency. The same equivalence principles apply in this case as well. Thus, truth tables, FSMs, BDDs, AIGs, Boolean circuit representations, *etc.*, are all valid Boolean models. And the Booleanization techniques covered in the next few chapters can produce many such different kinds of Boolean models.

3.3 Booleanization vs Discretization

Our usual experience while explaining to others the concept of Booleanization has been that people often tend to confuse Booleanization with discretization. Therefore, we decided to add a section to this chapter that is devoted to clarifying the key differences between Booleanization and discretization. While some of this has already explained above in Section 3.1, we believe that the additional examples and discussions in this section will provide further clarity. If the reader is confident that he already understands the important differences between discretization and Booleanization, he may safely skip this section.

Let us first start with discretization, which is an easier concept to understand than Booleanization. The discretization problem is stated as follows: given a continuous waveform (for example, an analog voltage/current that varies with time), discretization refers to the problem of approximating the given waveform using only a finite number of discrete quantized levels. Note that throughout this dissertation, when we use the term discretization, we quantize both time and space; it is also possible to discretize a waveform in space without discretizing it in time, but we will not do so in this dissertation.

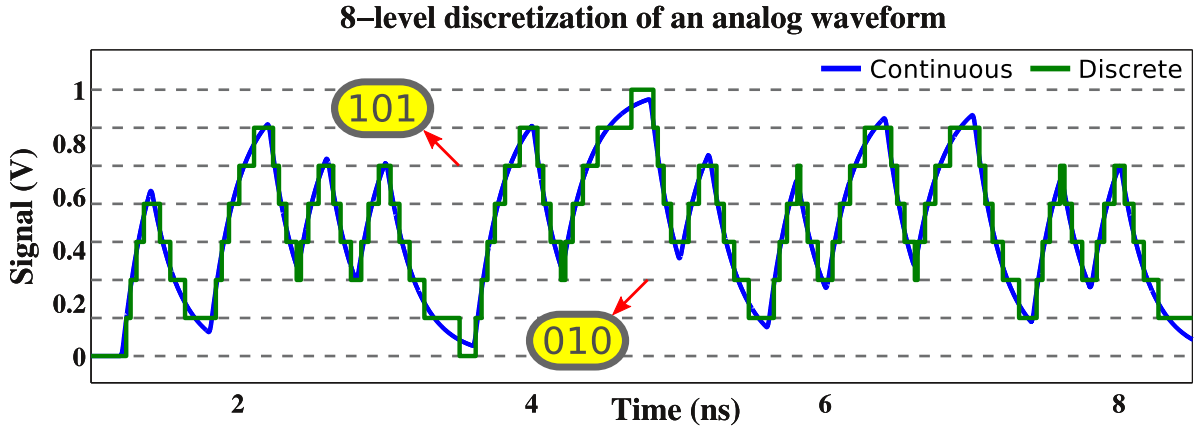


Figure 3.7: Discretization of a waveform.

Fig. 3.7 illustrates the concept of discretization. The blue waveform in the figure is an analog waveform. In this case, the waveform happens to be the output waveform produced by an RC chain for a *specific input waveform*.

The discretization is carried out by first quantizing the output waveform space into a finite number of discrete levels (in this case, 8 levels, placed uniformly apart). Then, each quantized level is assigned a *symbol* such as a bit vector encoding (in this case, the 8 levels are encoded in increasing order by using bit vectors from 000 through 111, as shown in Fig. 3.7). Finally, the horizontal axis (time) is discretized, and the given analog waveform is sampled and quantized to the closest discrete level at these time points. Thus, at every discrete time point, one obtains a discrete symbol or bit vector that represents an approximation to the analog waveform value at that time point. In this way, any analog waveform can be

represented, for example, as a sequence of bits. This completes the process of discretization. Indeed, discretization is conceptually very simple, and also not very hard to implement in practice (for example, a simple computer program for discretizing a waveform can be written in a few minutes). This problem is easy because, in the case of discretization, we only deal with *one specific waveform at a time* (as Fig. 3.7 shows).

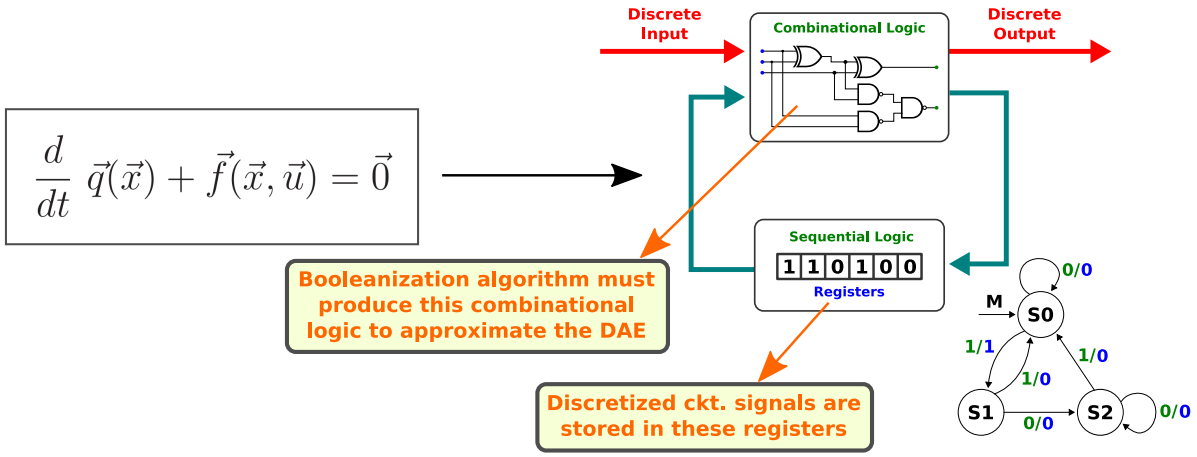


Figure 3.8: Booleanization of a DAE.

Booleanization (illustrated in Fig. 3.8), on the other hand, is a more difficult problem. In Booleanization, we are not given a single waveform. Rather, we are given a DAE (a system of differential-algebraic equations). The DAE implicitly encodes an *infinite number of input and output waveforms*, because for every input waveform, the DAE produces a corresponding output waveform. The Booleanization problem involves designing an FSM (represented, for example, in Boolean circuit form as shown in Fig. 3.8) that "mimics" the DAE. This means we want to construct an FSM that produces the correct sequence of bits at the output for *every conceivable input waveform* to the DAE. This is a much harder problem than discretization, which involves just producing the correct sequence of bits for a single waveform. To accomplish this for a real-world AMS system, the FSM that is constructed typically needs to (a) store a few discretized circuit variables (such as voltages and currents) in state registers, and (b) operate on and update these registers using a carefully designed combinational logic block whose objective is to approximate the given system's continuous-time behaviour. This is all illustrated in Fig. 3.8.

Thus, to summarize, discretization is a process that operates on single waveforms, while Booleanization operates on entire DAEs. Discretization is an easy (almost trivial) problem, but Booleanization is highly non-trivial. Discretization is often a small step that is carried out as part of the Booleanization process. The next section makes the connection between the original AMS system and its Booleanized version more precise.

3.4 The relationship between an AMS system and its Booleanized version

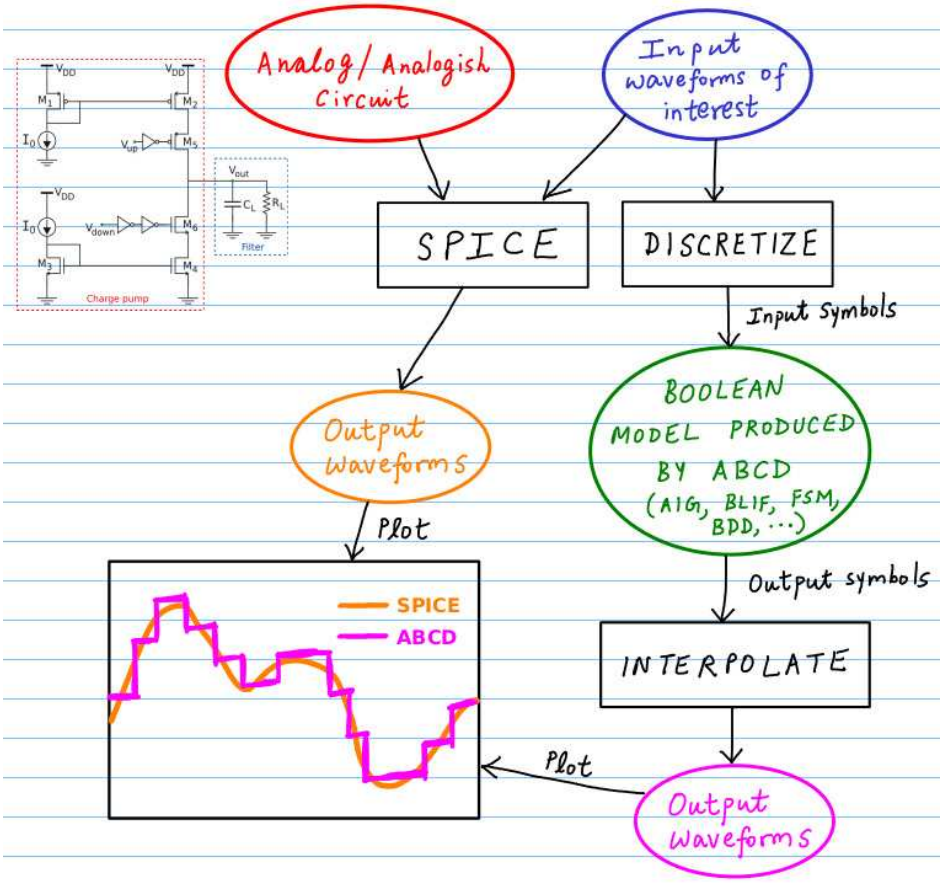


Figure 3.9: The precise relationship between an AMS system and its Booleanized version. Inputs to the AMS system, when discretized and fed to the Booleanized version, result in output symbols that when mapped back into the continuous domain closely match the continuous outputs produced by the original AMS system.

In this section, we try to draw a precise connection between an AMS system and its Booleanized version. Fig. 3.9 illustrates this connection.

Let us say we are given an AMS system S_A to Booleanize, either as a SPICE netlist or as a DAE system. For example, in Fig. 3.9, this system is a charge pump circuit expressed as a SPICE netlist. Furthermore, let us assume that we have run a Booleanization algorithm and obtained a Booleanized version S_B of this AMS system. Recall that S_B, by construction, will be a purely Boolean model.

The question we are concerned with is: what is the relationship between S_A and S_B? In

other words, given how S_A behaves, what conclusions can we draw about the behaviour of S_B?

Since S_B is intended to mimic the behaviour of S_A, one immediate conclusion is that both S_A and S_B will have the same number of inputs and outputs. One can think of S_B as a reduced order macromodel [40] of S_A, with the provision that since S_B is purely Boolean, its inputs and outputs will all be discrete, whereas those of S_A could be continuous.

This gives us a second insight: for S_B to be a good approximation to S_A, we need to be able to translate between the continuous domain of S_A's inputs and outputs to the discrete domain of S_B's inputs and outputs. That is, given a (continuous) input waveform for S_A, we must be able to discretize it into a sequence of symbols and feed this discrete symbol sequence to S_B. Similarly, given a sequence of output symbols produced by S_B, we should be able to interpolate this sequence and construct a continuous waveform that is comparable to the output waveform produced by S_A. This is also illustrated in Fig. 3.9.

Finally, we come to our most important requirement: what does it mean to say that S_B is a *good approximation* to S_A? For this, let us consider a continuous input waveform $u(t)$ to S_A. Let $y(t)$ be the output waveform (also continuous) produced by S_A for the input $u(t)$. We can determine $y(t)$, for example, by SPICE simulation.

The goal is to now ensure that S_B and S_A behave "similarly" on $u(t)$. To explore what this means, let us discretize $u(t)$ (in time and space) and obtain a sequence of *input symbols* $\{U_0,\ U_1,\ U_2,\ \ldots\}$ that we feed to the Booleanized version S_B. Let the corresponding output symbols produced by S_B be $\{Y_0,\ Y_1,\ Y_2,\ \ldots\}$. From the discussion above, we should be able to interpolate the sequence $\{Y_0,\ Y_1,\ Y_2,\ \ldots\}$ and map it back into a continuous waveform. Let us call this continuous waveform $\hat{y}(t)$. Clearly, for S_B to be a good approximation to S_A, $\hat{y}(t)$ should be a good approximation to the output $y(t)$ produced by S_A. This should hold true both "qualitatively" and "quantitatively".

By "qualitatively", we mean that the waveforms $y(t)$ and $\hat{y}(t)$ should "look like they have the same shape". For example, if $y(t)$ increases with time, $\hat{y}(t)$ should not decrease with time; if $y(t)$ switches back and forth between two values, $\hat{y}(t)$ should not settle down to a fixed point, and so on. It is easy to come up with naïve Booleanization procedures that result in dramatic qualitative differences between the Boolean model and SPICE. For example, Boolean models can have undesirable artifacts such as "fake fixed points" (Section 6.3.2). Therefore, while Booleanizing a system, one must take care to ensure that the Boolean model's predictions match SPICE qualitatively.

Assuming that $y(t)$ and $\hat{y}(t)$ are qualitatively similar, the next criterion for S_B to be a good approximation to S_A is that these waveforms should also match "quantitatively". This means that the absolute value of the difference between $y(t)$ and $\hat{y}(t)$ should be as small as possible, for every t. Of course, since $y(t)$ is continuous-valued and $\hat{y}(t)$ is discrete-valued (Fig. 3.9), we expect that there will be quantitative differences between the two. But we would like to minimize these as far as possible. In other words, we would like to construct the Boolean model in such a way that the output produced by *Booleanization* ($\hat{y}(t)$) closely matches the output that would be produced if we subjected $y(t)$ to *discretization* (using the same quantized levels).

Moreover, the "qualitative" and "quantitative" criteria above should hold true not just for a single input waveform $u(t)$, but for *every* input waveform (or at least, a large class of relevant input waveforms). This, indeed, is the precise goal of Booleanization, and this is the connection between an AMS system and its Booleanized version (as illustrated in Fig. 3.9).

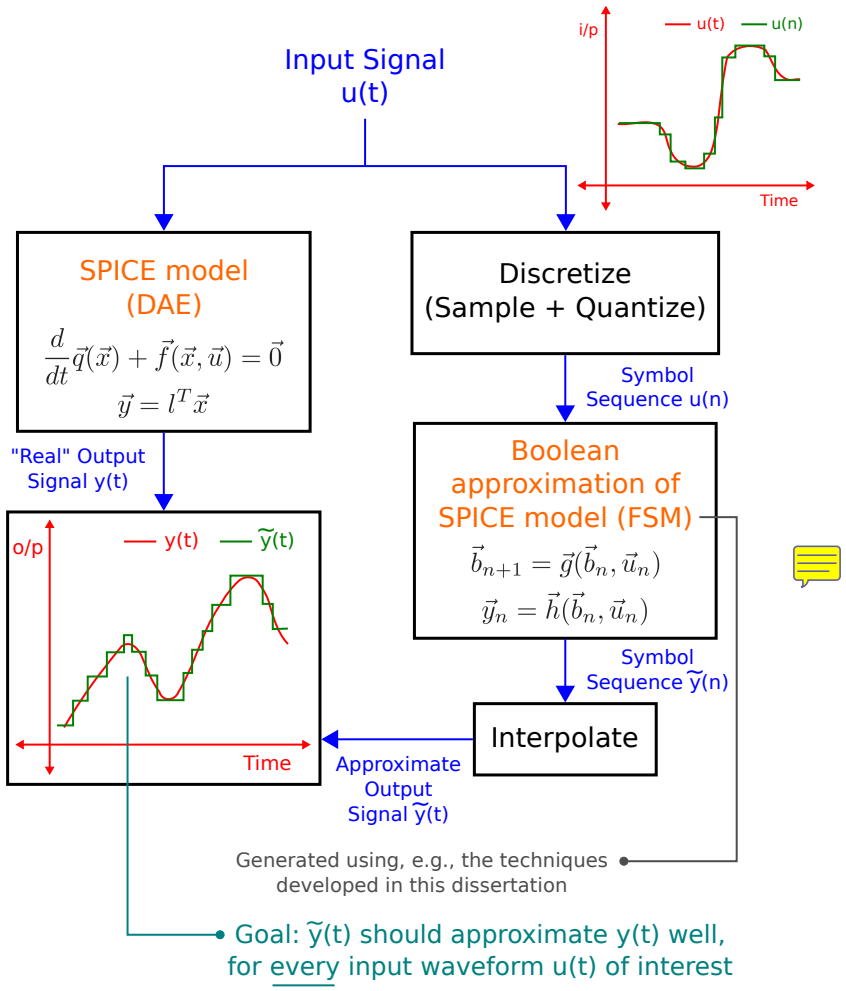


Figure 3.10: A more precise version of Fig. 3.9, illustrating what it means to approximate SPICE models via purely Boolean abstractions.

Fig. 3.10 illustrates these ideas more precisely. As the figure shows, we start with a SPICE-level model for the underlying AMS system or component. As mentioned above, this is a DAE system; it takes the following form:

$$\frac{d}{dt}\vec{q}(\vec{x}) + \vec{f}(\vec{x},\vec{u}) = \vec{0}, \text{ and}$$
$$\vec{y} = l^T\vec{x}, \tag{3.1}$$

where $\vec{u}$ denotes the inputs to the AMS system, $\vec{y}$ denotes its outputs, and $\vec{x}$ denotes its internal state space (typically a vector of voltages and currents).

Our Boolean approximation to the above system, by definition, can only take Boolean inputs. So we discretize (sample at a sufficiently high rate and quantize) the waveform $\vec{u}(t)$ above into a symbol sequence (a sequence of bit vectors) $\vec{u}(n)$ or $\{u_n\}$, which we feed as input to our Boolean model (as shown in Fig. 3.10), which behaves according to the following purely Boolean recurrence equations:

$$\begin{aligned} \vec{b}_{n+1} &= \vec{g}(\vec{b}_n, \vec{u}_n), \text{ and} \\ \vec{y}_n &= \vec{h}(\vec{b}_n, \vec{u}_n), \end{aligned} \tag{3.2}$$

where $\vec{b}_n$ denotes the (purely Boolean) state space of the system at time n, and other symbols follow similarly.

The output bits $\{\vec{y}_n\}$ produced by our Boolean model are then collected and interpolated, resulting in an *approximate output* waveform $\vec{\tilde{y}}(t)$. Our goal is to construct the Boolean model in such a way that, for *every* input waveform $\vec{u}(t)$ of interest, the approximate output $\vec{\tilde{y}}(t)$ produced by our Boolean model matches the output $\vec{y}(t)$ predicted by SPICE as closely as possible (with allowance for discretization errors and such, as shown in Fig. 3.10).

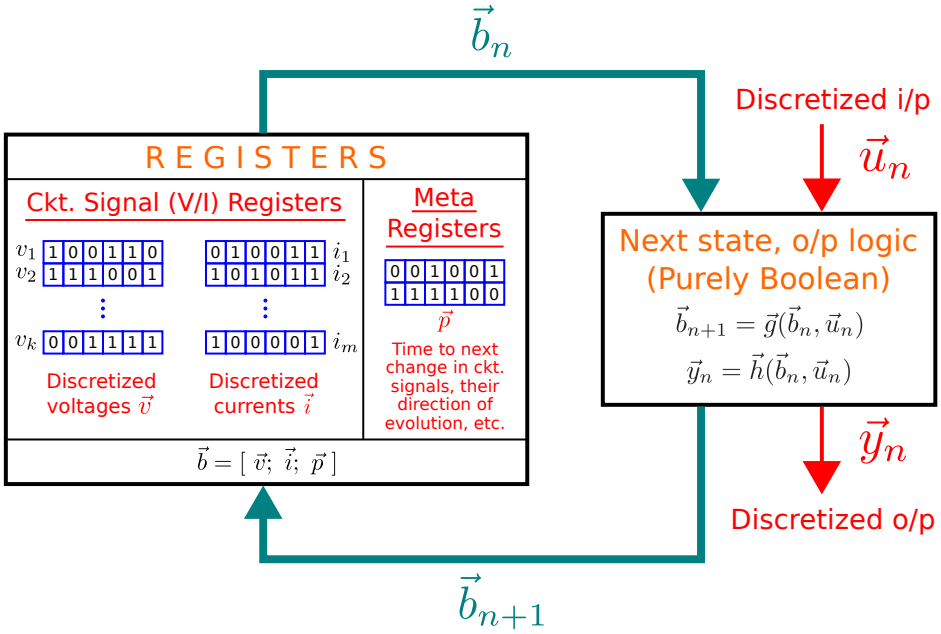


Figure 3.11: Structure of a purely Boolean model that approximates a SPICE-level DAE.

The algorithms and techniques described in the next few chapters all aim to realize this goal, for several different kinds of AMS systems. Fig. 3.11 shows the structure of

such a "Booleanized AMS" model. This works by maintaining an *internal Boolean state* (denoted $\vec{b}_n$) that is stored in *registers*. Each register bit stores either a binary 0 or a binary 1 at any given time. The registers are *clocked* at a frequency that is sufficiently high to capture all relevant SPICE-level dynamics of the given system. As Fig. 3.11 shows, a typical Booleanized AMS model contains two kinds of registers: (1) "circuit signal registers" that store discretized approximations of key voltages and currents in the circuit (where, unsurprisingly, the finer this discretization, the better the accuracy and fidelity of the Boolean model via-à-vis SPICE), and (2) "meta registers" that store information about how these signals are evolving in time (for example, whether these signals are increasing or decreasing, the number of clock cycles to the next discretized change in these signals, *etc.*).

Given the state of the registers $\vec{b}_n$ at time n, and the input $\vec{u}_n$ to the system at this time (which, from Fig. 3.10, is a discretization of the DAE input $\vec{u}(t)$), the Boolean model contains *logic* for updating the registers (*i.e.*, computing their *next state* $\vec{b}_{n+1}$), and for computing the output $\vec{y}_n$ (which, from Fig. 3.10, is designed to approximate the SPICE output $\vec{y}(t)$). This logic features *only* Boolean operations (AND, NAND, *etc.*); one can think of it as a combinational digital circuit. In this way, the SPICE-level behaviour of an AMS system can be approximated in Boolean form.

3.5 Why should Booleanization even be possible?

We acknowledge that the whole idea of Booleanizing analog systems, even after the explanations and descriptions of the previous sections, can still seem foreign and outlandish to a reader who is just getting acquainted with these notions. After all, continuous dynamical systems such as DAEs are completely different mathematical entities compared to FSMs and other Boolean models. So, at a fundamental level, why should there be a connection between these two kinds of systems, which, at the surface, seem poles apart from one another? In other words, is there intuition to support the idea that Booleanization of practical analog systems is even possible, let alone practical?

We approach this question from two different angles.

States and transitions. Both DAEs and FSMs have state spaces, and transitions within state spaces. Thus, if one were to draw a connection between DAEs and FSMs, the core ideas of "states" and "transitions between states" might be as good a place to start as any. At its core, a DAE simply specifies how a state vector evolves over time. That is, if the current state of the DAE is known, and the current inputs to the DAE are known, the equations that make up the DAE simply specify how the future state of the DAE (as well as the DAE's outputs) will evolve as a function of time. Thus, if the current state and the current inputs to the DAE are known, then the state of the DAE at a short time into the future can be predicted with reasonable accuracy, for most DAEs. This is a fundamental idea that underpins many mathematical concepts, including Taylor series approximations, Forward Euler integration [7], *etc.*. The idea of "states", and "transitions between states", also lends itself to discretization. For example, what if we discretize the

continuous state, input, and output spaces, as well as time, extremely finely? Now, we have a discrete (albeit very finely discretized) state space, and we can conceive of a non-deterministic FSM constructed on this discrete state space. In this FSM, an arc labelled with the input symbol i is drawn from the discrete state u to the discrete state v if and only if v is reachable from u in one unit of (discrete) time, while the system's discretized input remains i throughout this time. This FSM is non-deterministic by construction, but it is reasonable to believe that, in the limit that the fineness of discretization of the various spaces (and time) goes to infinity, the FSM begins to resemble the DAE more and more, until it eventually becomes equivalent to the DAE at the limit. Thus, our thought experiment above yields a sequence of non-deterministic FSMs that converges to a given DAE. Thus, it seems reasonable from this argument that one can approximate a DAE using a reasonably finely discretized non-deterministic FSM, which provides some intuition into why Booleanization should be possible, at least for well-behaved DAEs that satisfy existence and uniqueness properties, that feature reasonably smooth waveforms, that are robustly stable and non-chaotic, *etc.*.

SPICE, at its core, is purely "Boolean" and deterministic. Our second approach to this question involves considering the environment that SPICE runs in. Typically, SPICE simulations are carried out using microprocessors that use either 32 or 64 bit computations. That is, all the "continuous" voltages and currents that are simulated by SPICE are in fact represented internally using 32 or 64 bits each, and all operations carried out by SPICE can be viewed as purely Boolean operations on these bit vectors (using purely Boolean hardware such as adders, multipliers, floating-point units, *etc.*). Indeed, viewed this way, any computer program that occupies only a finite amount of memory is in fact a deterministic FSM. Thus, a SPICE simulation of a circuit can be thought of as simply a deterministic FSM simulation. The state of this FSM consists of the values stored in all the registers/memory locations occupied by SPICE. And the state transition relationship of this FSM is simply the logic followed by SPICE (translated into purely Boolean assembly instructions and logic operations on registers and/or other elements of the state space by a compiler). Therefore, since SPICE simulations are widely accepted as the golden standard of accuracy, it is clear that if one is willing to use very large FSM state spaces, ALUs, FPUs, *etc.*, one can approximate most analog circuits to SPICE-level accuracy using deterministic FSMs (with suitable Boolean encoding of state transition operations). In other words, SPICE itself represents a 32 or 64 bit Booleanization of the underlying circuit DAE – which is evidence that Booleanization is in fact possible.

Of course, the arguments above are merely thought experiments to justify why Booleanization of DAEs may be possible. They do not represent practical ways of going about Booleanizing DAEs. But having provided some intuition to show that the idea of Booleanization, as strange as it seems at first glance, is in fact possible, we are now in a position to go ahead and look at an example of how Booleanization works in practice.

3.6 A simple example: Booleanizing an RLGC filter

Having discussed the "theory" and core concepts behind Booleanization, we now provide an illustration of Booleanization in action, using as example an RLGC chain, Booleanized using ABCD-L (covered in Chapter 5).

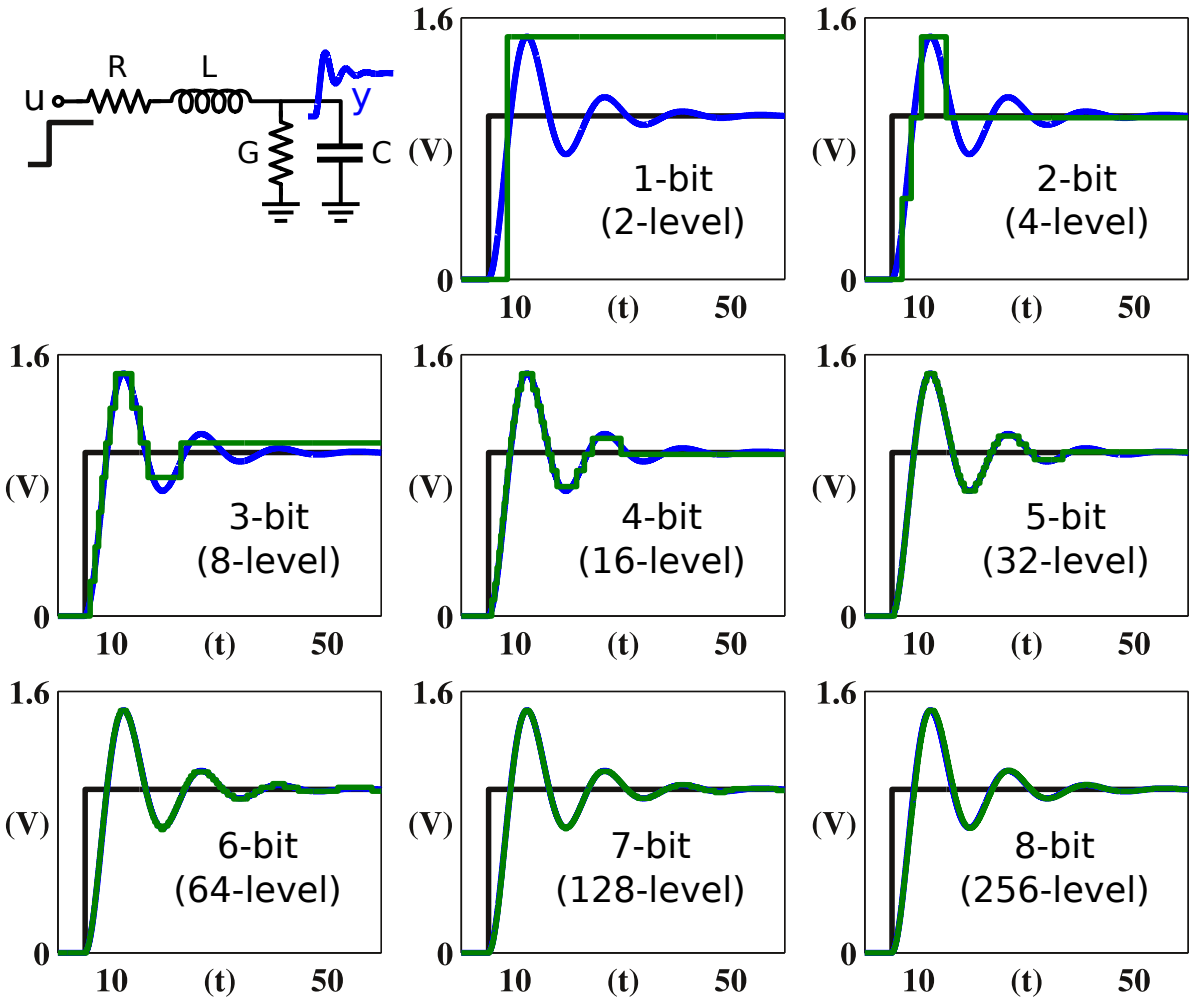


Figure 3.12: Booleanization of an RLGC filter using ABCD-L, resulting in Boolean models of progressively higher accuracy (and, unfortunately, progressively increasing model sizes, as measured by the number of logic gates, flip-flops, *etc.* in the Boolean model description). The filter's response to a step input $u(t)$ (in black) is computed analytically (the waveforms in blue), and this waveform is compared against the waveforms predicted by the Boolean models generated by ABCD-L (the waveforms in green). In the plots above, the X-axis denotes time in RC units, and the Y-axis denotes voltages in Volts.

Fig. 3.12 presents the results of this Booleanization. The circuit (an RLGC filter) is

shown at the top left of the figure. This is a linear system of size 2, whose eigenvalues are both complex. The DAE system representing this circuit takes the following form:

$$\begin{pmatrix} C & 0 \\ 0 & L/R \end{pmatrix} \frac{d}{dt} \begin{pmatrix} y \\ i_L \end{pmatrix} + \begin{pmatrix} G & -1 \\ 1/R & 1 \end{pmatrix} \begin{pmatrix} y \\ i_L \end{pmatrix} + \begin{pmatrix} 0 \\ -1/R \end{pmatrix} u(t) = \vec{0}, \tag{3.3}$$

where $u(t)$ is the input to the filter, $y(t)$ is the filter's output, and $i_L(t)$ is the time-varying current flowing through the inductor from left to right in the circuit.

As described in Section 3.1, the first step in the Booleanization process is discretization. Here, ABCD-L discretizes the input waveform u, the internal current i_L, and the output y using bit vectors of length m, where higher values of m correspond to finer quantization of the underlying analog signals. The next step, as described in Section 3.1, is to Booleanize the DAE system of equations above, which is done automatically by ABCD-L (please see Chapter 5 for a detailed explanation of how ABCD-L works). The figure above shows that, as we increase the number of quantization bits m from 1 to 8, the responses predicted by ABCD-L's Boolean model (the green waveforms) gradually match the actual system's response (the blue waveforms) more and more accurately (please see the discussion in Section 3.4 for a detailed description of how these waveforms are mapped between discrete and continuous domains), duplicating important features such as overshoot and ringing. While we have taken a step input as the example in Fig. 3.12, the Boolean models produced by ABCD-L produce similar results for virtually every input waveform of interest.

3.7 The tradeoff between accuracy and Boolean model size

The example above points us to an interesting observation: in spite of using only purely Boolean operations and abstractions, Booleanization algorithms such as ABCD-L are in fact able to reproduce the continuous-time dynamics of AMS systems with *high accuracy*. Furthermore, by increasing the number of bits used to discretize/represent the underlying circuit signals, this accuracy can often be made as high as desired.

This raises an important question: are we giving up anything in exchange for this high accuracy? In other words, is there a flip side to increasing the number of bits used to discretize the underlying circuit signals?

Unfortunately, the answer is yes. As we increase the number of bits used to represent the underlying circuit signals, the Boolean models produced by techniques like ABCD-L tend to grow in size. That is, the greater the number of bits used, the greater the size and complexity of the resulting Boolean model, resulting in a larger CPU/memory footprint when these models are ultimately used for high-speed simulation or AMS verification. Also, as the number of bits increases, the Booleanization algorithm itself runs slower, requiring more time to generate the Boolean model (and more space to store it).

Furthermore, the size of the Boolean model also depends on the fineness of discretizing time. Recall that the Boolean model is clocked by discretizing time. As the period of discretization becomes smaller and smaller, the Boolean model tends to become larger – simply because a larger state space is required to model the longer "sequential memory" of the Boolean model. However, as we discretize time more finely, the Boolean model also becomes more accurate because it now has greater timescale resolution. In practice, the Boolean model is clocked at a frequency that is slightly higher than the rate at which the underlying AMS system's continuous-time dynamics evolve. For example, if the AMS system is a charge pump that has a peak-to-peak delay of a few nanoseconds, we might use a 100ps clock for the Boolean model that approximates the charge-pump dynamics.

Thus, there exists a tradeoff between Boolean model size and accuracy. Model size and accuracy both typically increase with the number of bits/levels used for discretizing the underlying circuit signals. Model size and accuracy also increase when time is resolved more finely (*i.e.*, when a faster clock is used for generating the Boolean model).

Therefore, while Booleanizing AMS systems, intelligent choices need to be made regarding the fineness of discretization used – both in space and in time. Typically, one starts with a reasonably small number of bits (like 4 or 5) per circuit signal, and a reasonably coarse clock period (such as one-fifth to one-tenth of typical circuit response times, or 5 to 10 times the circuit's "bandwidth"), and gradually increases these until a Boolean model of sufficiently high accuracy is obtained.

3.8 Why Booleanize AMS designs?

The previous sections introduced the key concepts behind Booleanization, provided examples, *etc.*. We now want to change gears a little bit and discuss the motivation for Booleanization, *i.e.*, why would we want to Booleanize AMS systems and how could Booleanization help solve the problem of bug-free AMS design? While some of this has already been covered in bits and pieces in previous chapters and elsewhere in this chapter, we believe that this section will provide a more complete picture and a fuller discussion of the many potential benefits of Booleanization for the AMS design problem.

We can identify 3 important reasons for Booleanizing AMS systems: (1) to enable all-Boolean formal verification of AMS designs, (2) to enable high-speed simulation of AMS designs, and (3) to work in conjunction with hybrid system methods and frameworks, and help them scale to real-world AMS designs by replacing some of their time-consuming continuous variables with purely Boolean ones instead. We discuss these points individually below.

3.8.1 All Boolean formal AMS verification

We believe that Booleanizing AMS designs (for example, using ABCD) offers several compelling advantages from the point of view of both accuracy and scalability.

Indeed, because Booleanization in general (and ABCD in particular) produces purely Boolean models, it is well-suited for use in conjunction with existing techniques for Boolean reachability analysis and formal verification. By reducing continuous-time dynamics to Boolean form, we believe that Booleanization techniques like ABCD make it possible to leverage powerful Boolean engines (*e.g.*, open-source software like ABC, or commercial verification tools like Conformal) already developed for model checking/reachability analysis of digital designs, and repurpose the same tools for the analysis and verification of AMS designs as well. This is illustrated in Fig. 3.13, using a SAR-ADC as an example (for more details on how the SAR-ADC works and why it poses challenges for AMS designers, please see Section 1.3.1).

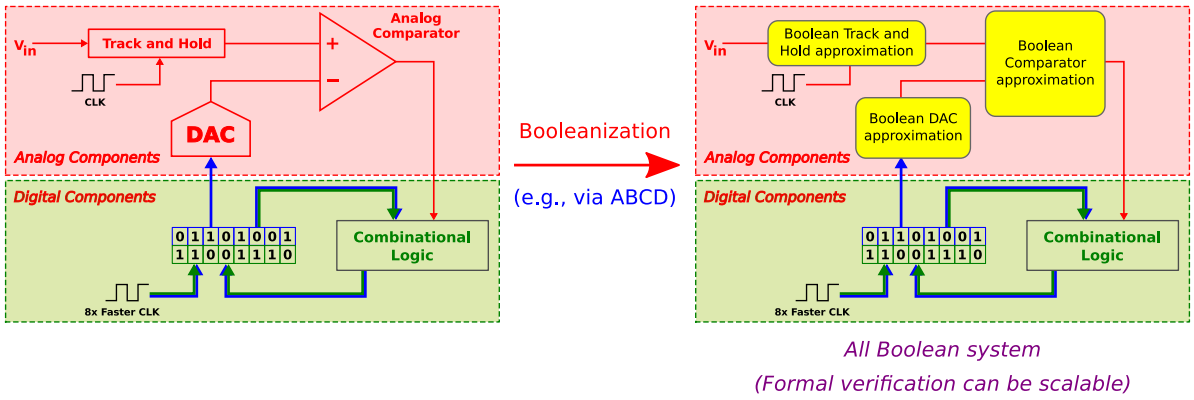


Figure 3.13: Booleanization of the components of a SAR-ADC can result in an all-Boolean system that can then be property-checked using existing powerful Boolean engines like ABC.

Moreover, because ABCD-generated models are learned from SPICE simulations, they can model the underlying AMS system's behaviour with a high degree of accuracy that is usually not possible with simplified behavioural models typically used for AMS verification. This ability to model features of SPICE-level dynamics can greatly increase the applicability and appeal of Booleanization-based verification tools and flows for practical designs such as the systems described in Section 1.3, including SAR-ADCs, SERDES systems and PLLs, high-speed communication sub-systems, *etc.*. In this way, we believe that Booleanization and ABCD will eventually help bridge the gap between SPICE-level detail and the models used by the current generation of AMS verification engines.

Furthermore, we believe that Booleanization opens the door to more scalable formal verification of AMS designs because purely Boolean analysis/verification tools tend to be able to scale to much larger designs than AMS verification tools that sacrifice scalability for the ability to model continuous variables.

3.8.2 High-speed AMS simulation

It is well-known that purely Boolean systems are much easier and faster to simulate than continuous DAE systems and hybrid systems. Thus, the Boolean models produced by Booleanization techniques like ABCD can be simulated very efficiently.

This is because such simulations are all carried out at the logic level, in terms of discrete symbol sequences rather than operations on differential equations. Indeed, the time-domain "transient" simulation of ABCD-generated models involves just a single *constant-time* memory look-up (of the underlying FSM state transition table) per time-point. There is no need to evaluate complicated device models or solve differential equations via linear multi-step integration methods [41], Newton-Raphson iterations, *etc.*, as is typical for DAE/SPICE-like simulation of AMS designs.

Therefore, in many applications, ABCD can be used as a much faster, reasonably accurate, drop-in replacement for SPICE. This has the potential to significantly speed-up system-level simulations for AMS design.

3.8.3 Combination with hybrid system methods and frameworks

Finally, in the context of AMS verification via hybrid system methods and frameworks, we know that existing hybrid system approaches are unable to cope with more than a few continuous variables, whereas they can comfortably handle hundreds, if not thousands, of purely Boolean variables. Therefore, in the interest of scalability, it is desirable to use continuous quantities as sparingly as possible (because they are often computationally orders of magnitude more expensive), and to model most kinds of dynamics using purely Boolean quantities (which are computationally much cheaper) instead. We believe that Booleanization and ABCD can offer powerful advantages here. For instance, by taking advantage of ABCD's ability to model SPICE-level dynamics in purely Boolean form, we believe that much of an AMS system's behaviour will be representable using "cheap" purely discrete/Boolean variables, which frees up "precious" continuous variables for use only when absolutely necessary. This has the potential to help arrest the scalability issues inherent to existing hybrid system approaches. Therefore, with ABCD, it may be possible to expand the scope of hybrid system approaches to much larger systems than they can handle at present. Although this notion has been theoretically studied before (*e.g.*, see [42]), we believe that ABCD constitutes the first practical solution for Booleanizing linear and non-linear SPICE-level continuous-time dynamics in an accurate and systematic manner.

3.9 The need for automated Booleanization

The last few sections presented key Booleanization concepts and provided motivations for Booleanizing AMS designs. This covered the "what" and the "why" of Booleanization, while leaving the "how" of Booleanization (*i.e.*, the techniques and procedures to use for Booleanizing AMS designs) largely unanswered.

In this section, we would like to briefly touch upon the "how" of Booleanization, by drawing a broad distinction between *manual* Booleanization and *automated* Booleanization. We will make the case that, while manual Booleanization is a valuable skill to possess and should not be lightly dismissed, there is a distinct advantage to automating the process of Booleanization and developing tools, techniques, and algorithms for automated Booleanization.

Broadly speaking, the process of Booleanization can be carried out either in a manual fashion or in an automated way. Manual Booleanization of an AMS system involves studying the system, deciding on the key factors and behaviours to be modelled, and then constructing a Boolean model that accurately reflects all these behaviours. This carries some obvious limitations. For example, manually Booleanizing a SERDES system would require deep knowledge of and insight into how the system works and the various non-idealities that are likely to exert a meaningful impact on its performance. Such design knowledge is typically the province of only a select few. Furthermore, manual Booleanization often involves making *ad hoc* modelling simplifications; for example, deciding on what factors to model always carries the risk that some key performance-limiting factors are ignored or not fully taken into account. For example, manually developed models are unlikely to capture non-ideal effects and system behaviours that the model developer is either unaware of, or incorrectly considers not important enough to watch out for. Also, manual Booleanization tends to be tedious and error-prone, and not particularly suitable in situations where the underlying design is being modified often.

However, in many situations, manual Booleanization can be beneficial as well. This is because the models produced by manual Booleanization are often simpler, more intuitive, and easier to understand than Boolean models generated automatically by running an algorithm. In most cases, such models are also more efficient from a simulation and verification perspective, because they tend to be much simpler than auto-generated models. Indeed, for some AMS verification problems, auto-generated models can even be computationally infeasible to verify, while manually generated models can be formally verified with relatively little time and trouble.

Notwithstanding the above, there is a strong case to be made for automated Booleanization. Ideally, automated Booleanization of an AMS design would work as follows: it would take as input a SPICE netlist (or similar description) of an AMS component plus a few Booleanization parameters (such as the fineness of discretization to use, the Booleanization clock period, *etc.*), and at the push of a button, it would produce as output a Boolean model for the given AMS system. This, of course, has a strong appeal: because Boolean models are derived from SPICE-level (or similar) models, a large number of subtle effects can be considered and accurately modelled. When done automatically, this is especially attractive because, in modern technologies, even the most talented designer will typically miss a variety of important effects captured by SPICE. Furthermore, because the process of deriving Boolean models is automated, it can be carried out in a systematic way without giving in to pre-conceived biasses and missing important corner cases. In other words, the high reliability provided by automation helps ensure the quality and completeness of the Boolean models

generated. Moreover, automated Booleanization eliminates the need for the end user to have a detailed knowledge and understanding of how the underlying AMS design works at the lowest-level of detail. This frees up the end user to work at a higher level, concentrating on system-level details and performance without getting bogged down in the complexity of individual AMS components, while being rest assured that the automated Booleanization algorithms will account for all important factors that affect system-level performance.

3.10 Introducing ABCD

Based on the discussion above, we believe that there is a strong need for developing automated Booleanization techniques and algorithms that come as close as possible to the "push button style Booleanization" described above.

To this end, we have developed ABCD, a suite of automated Booleanization tools, techniques, and algorithms that are designed to work for a wide variety of AMS designs. The next few chapters of this dissertation cover some of these tools, techniques, and algorithms.

Chapter 4, for example, introduces DAE2FSM, an Angluin-style[2] FSM abstraction technique that applies computational learning algorithms to SPICE-level AMS designs of "digitalish" systems (systems that are intended to function as digital logic blocks, but in practice exhibit pronounced performance-limiting analog traits and characteristics). Next, Chapter 5 covers ABCD-L, a technique that we developed for automatically Booleanizing Linear Time Invariant (LTI) systems based on eigen-analysis. Finally, Chapter 6 covers ABCD-NL, a technique designed to Booleanize a large class of genuinely non-linear AMS designs by separating their behaviour into "DC" dynamics and "transient" dynamics.

These techniques (namely, DAE2FSM, ABCD-L, and ABCD-NL) are all part of the ABCD umbrella of Booleanization techniques. While the core technical ideas and algorithms underlying these techniques are very different from one another, they do share some common features; for example, these techniques all work by discretizing the signals in the underlying AMS circuit (*e.g.*, the circuit's inputs and outputs, and perhaps some internal signals as well). The clock period for the Booleanized model and the number of bits to use for this discretization can both be specified by the user; as discussed in Section 3.7 above, these are usually decided while keeping in mind the tradeoff between accuracy and Boolean model size. Indeed, ABCD offers the user fine-grained control over almost every aspect of the discretization, including the thresholds used for discretization; these thresholds become important if the user requires non-uniform discretization of the circuit's underlying voltages and currents, as opposed to uniform discretization which is the default.

Once the discretization parameters are fixed, ABCD typically carries out some analysis of the underlying AMS design. This analysis may involve, for example, running carefully chosen SPICE simulations of the design. Or, if one knows that the design satisfies some special properties (for example, if one knows that the design is linear and time-invariant),

[2]This is a reference to the well-known black-box algorithm for learning deterministic finite automata from simulation traces, published by Dana Angluin in 1987 [43]. For more details, please see Chapter 4.

this analysis phase may be able to take advantage of this information. The eigen-analysis carried out by ABCD-L is a good example of this kind of specialized analysis.

Based on the analysis above, ABCD then produces an FSM approximation of the underlying AMS design. This FSM's input and output alphabets are determined by the discretization parameters chosen by the user for the circuit's input and output signals. For example, if the user decides to discretize the circuit's inputs using 3 bits, the FSM's input alphabet will contain 2^3, or 8, symbols.

As with any FSM, the FSM created above specifies a set of *transitions*. These may be maintained in graph form, or table form, or encoded in Boolean circuit form, as discussed in Section 3.2. The important idea is that, given the *current* discrete state of the FSM, and a discretized version of the circuit inputs at each clock instant, the FSM description includes all the "logic" necessary to compute its *next* discrete state (*i.e.*, its state one clock period later), as well as a discrete approximation to the AMS system's outputs. This "logic" makes use of *only* Boolean logic operations (*e.g.*, `AND`, `OR`, `NAND`, `XOR`, *etc.*). This logic is typically learned based on the analysis phase described above.

In the end, even though ABCD approximates continuous systems in an all-Boolean fashion, the accuracy of ABCD-generated models tends to be much better than that of behavioural models typically used for AMS verification, because the analysis phase can be made to take into account SPICE-level details with high accuracy. Also, in principle, for well-behaved and stable systems, this accuracy can be made as high as necessary simply by increasing the number of bits (and the clock frequency) used to discretize the circuit's voltages and currents (for example, please see Fig. 3.12). Furthermore, since ABCD-generated models are all-Boolean, the scalability of verification involving these models is also likely to be better than continuous-time behavioural models.

Finally, ABCD also has some support for exporting the generated Boolean models into formats that existing Boolean verification tools (*e.g.*, ABC) directly accept as input (*e.g.*, PLA, BLIF, or AIG file formats). Thus, the Boolean models produced by ABCD are well-suited for use with cutting-edge formal verification/model checking engines. To the best of our knowledge, ABCD is the first suite of tools that offers the capability to transform a SPICE-level netlist of an AMS design into a purely Boolean "executable" model suitable for verification at the push of a button.

Over time, and with the development of additional algorithms and heuristics, we hope to eventually reach a stage where ABCD becomes a powerful industry-scale push-button solution for automatically deriving Boolean models for virtually any AMS design. We view this dissertation as a step in this direction.

Chapter 4

DAE2FSM: Automated Booleanization of "Digitalish" Systems

This chapter presents DAE2FSM, the first of three automated Booleanization techniques we have developed as part of the ABCD suite.

DAE2FSM works well for what we call "digitalish" designs: these are designs whose *intended* end-to-end functionality is purely digital, but whose *implementation* in practice tends to introduce analog non-idealities that significantly impact functional correctness as well as performance.

In this chapter, we start by modelling such designs with high accuracy as continuous-time dynamical systems (*e.g.*, full SPICE-level netlists specifying detailed transistor-level implementations of combinational and sequential logic modules). We then develop an algorithm to abstract the end-to-end I/O functionality of these designs (including the analog non-idealities exhibited by these designs) as purely Boolean Finite State Machines (FSMs).

This enables efficient simulation of large designs implemented with less-than-perfect devices and components, and also opens the door to formal verification of transistor-level designs against higher-level specifications. In particular, our automatically generated FSMs faithfully capture the behaviour of latches, flip-flops, and circuits constructed from them. Among other technical advances, we generalize an existing (binary-only) FSM-learning approach to arbitrary I/O alphabets, which empowers this approach to learn high-fidelity abstractions of multi-level-discretized, multi-input/multi-output systems.

The above approach (dubbed DAE2FSM), when applied to correctly functioning latches and flip-flops, is able to learn compact, multi-input FSM abstractions whose predictions closely match SPICE simulations. In addition, we have also applied DAE2FSM to produce multi-level-discretized FSM representations of "digitalish" systems that exhibit "analogish" traits, such as an overclocked, error-prone D-flip-flop. For such circuits, the automatically learned FSM abstraction includes additional states that characterise "failure modes" of the circuit for specific input sequences (these failure modes are also confirmed by SPICE simu-

lations).

Finally, we demonstrate that DAE2FSM is also applicable to larger and more complex multi-input, multi-output systems; for example, we are able to automatically derive an accurate FSM abstraction of a 280-transistor 0-to-5 increment/decrement counter, where all transistors were modelled as full BSIM4 [44] devices with hundreds of parameters each.

For a more succinct treatment of the material in this chapter, we refer the reader to our paper on DAE2FSM [45].

4.1 The need for DAE2FSM

With CMOS technology scaling to 22nm and below, individual devices (transistors), as well as digital building blocks (such as logic gates and flip-flops), are becoming increasingly "non-ideal". This compromises the clean Boolean abstractions (such as simple truth tables and idealized FSM descriptions) that are in heavy use even today to model, think about, and reason about the behaviour of digital designs.

Such clean Boolean abstractions underpin the effectiveness and power of the digital design paradigm. They are the abstractions that lie at the very core of a vast body of work on digital design and digital CAD/EDA tools. For example, a variety of algorithms for timing analysis, technology mapping, high-speed simulation, and formal verification of digital designs have come to rely on such clean Boolean abstractions.

Unfortunately, many components in cutting-edge digital systems today behave more like analog/RF circuits than like digital ones. The design, validation and debugging of digital systems with such components can be challenging because "analog issues" stemming from such nonlinear analog dynamics, non-ideal analog waveform shapes (that no longer match the ideal switching behaviours typically assumed by digital design tools), noise/interference, *etc.*, compounded by increased variability, cannot be directly captured within the clean Boolean modelling abstractions and simple delay frameworks that are natural for thinking about, modelling, and formally verifying the design of digital systems.

An alternative, of course, is to model such systems as full-fledged continuous dynamical systems at SPICE-level detail (or close to SPICE-level detail), using accurate transistor models (such as BSIM or PSP) that capture all the analog effects mentioned above. However, such purely SPICE-level approaches are impractical for two reasons. Firstly, SPICE-level (and similar) models tend to be suited mainly for simulation. Many other kinds of analysis algorithms developed for digital design, such as formal verification algorithms, timing analysis algorithms, *etc.*, typically do not work with SPICE-level (or even any kind of continuous) models. Secondly, even the simulation-based approaches (which SPICE-level models are amenable to) for validation and debugging are often completely impractical, on account of the large sizes of typical digital systems. For example, in existing digital design methodologies and flows, while components in cell libraries are simulated extensively at the SPICE-level, over a range of PVT corners, to verify functionality and to characterise delays, this process often takes several weeks or even months, and large digital designs are well outside the

scope of even the most capable SPICE simulators available today. Moreover, SPICE-level characterization cannot provide executable Boolean abstractions (such as FSMs) that can reproduce details of analog waveform shapes, rendering most digital CAD tools (*e.g.*, formal verification algorithms) inapplicable. Furthermore, such pre-characterization approaches are usually not well-suited for components whose functionality is affected significantly by complex analog effects that propagate through a large digital design; instead, they work well only for designs where the analog effects add up in a simple way (such as simple addition of delays, for example).

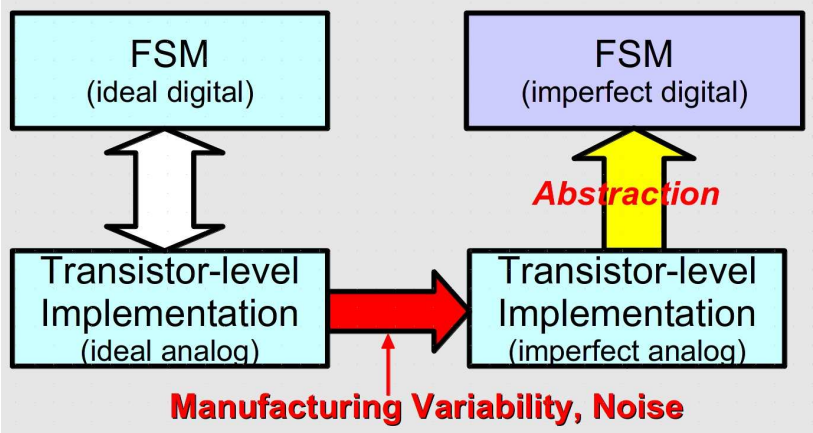


Figure 4.1: DAE2FSM: Transistor level non-idealities are captured in FSM representations.

With a view to addressing the issues above, we develop and demonstrate a technique (dubbed DAE2FSM[1]) to abstract executable Boolean descriptions of digital designs represented as full transistor-level circuits in SPICE-level detail. The central notion of DAE2FSM is to approximate such transistor-level circuits *accurately* as finite state machines (FSMs) by adapting and extending black-box computational learning techniques, such as Angluin's algorithm [43] for deriving deterministic finite automaton (DFA) abstractions from simulation traces. The resulting FSMs can not only capture the intended logical functionality of the circuit being abstracted, but can also take into account analog effects and non-idealities, producing "non-ideal FSMs" that accurately reflect actual (rather than intended) circuit operation. This overall concept is illustrated in Fig. 4.1.

In this chapter, we develop and apply an Angluin-based [43] computational learning technique that uses *finite alphabets of* $N \geq 2$ *symbols*, thus moving beyond the binary symbols used in prior work [1, 2]. This enables us to learn FSMs for transistor-level circuits with *multiple inputs*, by encoding input value or transition combinations using multiple symbols. Multi-symbol learning also enables us to obtain increased fidelity by using *multi-level discretizations* to approximate analog signals better. We also show how *non-deterministic FSMs* can be used to capture situations with unpredictable inputs or unpredictable circuit

[1] Differential-Algebraic Equation to Finite State Machine.

responses. We demonstrate the application of these advances on transistor-level latch and flip-flop circuits represented in full SPICE-level detail using BSIM transistor models [44], as well as on a counter circuit composed of several sequential and combinational components (also represented in SPICE-level detail).

We believe that DAE2FSM features several points of novelty and promise. For example, FSMs generated by DAE2FSM can be simulated (in discrete time in the logical domain) much faster than the underlying SPICE-level representations they are derived from, while at the same time capturing the impact of analog/manufacturing imperfections. Indeed, the results of logical FSM simulation can be translated back to analog values that, in many cases, reproduce SPICE-simulated waveforms well. Compromised functionality and failure modes are also captured by the FSMs, which opens possibilities for system-level/post-silicon workarounds.

Another important feature of DAE2FSM is that because it is a black-box technique that simply learns FSMs from simulation traces, it is suitable for application within simulation environments of the user's choice, thereby ensuring that no analog subtlety that the user's favourite simulator can predict is ignored in the generated FSM.

Also, the I/O-based FSM learning approach behind DAE2FSM ensures that only those details of transistor-level blocks that are "observable from the outside" (*i.e.*, relevant from an end-to-end I/O functionality or system perspective) are captured; in other words, only what is needed is represented in the learned FSM, and internal dynamics that do not have external effects are ignored.

Furthermore, the automated, push-button nature of FSM generation frees the user from having to understand in detail how a given transistor-level block functions; all that is needed is a SPICE-level (or similar) netlist that simulates.

We think that, from the standpoint of fitting into established design flows, the fact that simulation, validation and debugging can all be performed purely in the logical domain, using a variety of existing digital CAD tools that already exist for these purposes, is very attractive. For example, as described in Section 3.8, we believe that DAE2FSM opens the door to the application of Boolean formal verification and model checking techniques (*e.g.*, [6, 46]) for determining whether a given transistor-level component with strong analog characteristics satisfies a given property. Also, as described in Section 2.5, unlike alternative formal approaches based on hybrid system methods and frameworks (*e.g.*, [42, 47–49]), the models produced by DAE2FSM, by virtue of being purely Boolean, tend to suffer from fewer scalability limitations. Again, as mentioned in Section 2.5, another key advantage of DAE2FSM over such hybrid system methods is that no *a priori* modelling simplifications are required, since full SPICE-level transistor blocks can be directly abstracted as FSMs. This reduces the chances of false negatives (bugs that exist in the design but are not discovered because of *a priori* modelling similifications) during formal verification.

The rest of this chapter is organized as follows. In Section 4.2, we outline the multi-symbol FSM learning technique underlying DAE2FSM. As mentioned above, this is an adaptation of Angluin's well-known black-box DFA learning algorithm from computational learning theory. In the next few sections, we present detailed results obtained by applying

DAE2FSM to latch and flip-flop circuits, generating FSMs for single and multiple inputs, binary and multi-level discretizations, and properly functioning as well as failing circuits. We also demonstrate how DAE2FSM captures the intended functionality of a 280-transistor, 0-to-5 increment/decrement counter from a SPICE-level description. Finally, in Section 4.8, we discuss some important limitations of DAE2FSM; for example, we discuss why DAE2FSM works well for Booleanizing "digitalish" systems with analog non-idealities, but can fail badly when applied to (even simple) genuinely analog systems. Many of these limitations arise as a direct consequence of the black-box Teacher/Learner approach at the core of DAE2FSM (Section 4.2); understanding these limitations inspired us to overcome them by developing non-black-box Booleanization techniques like ABCD-L and ABCD-NL (covered in the next two chapters).

4.2 Core technique: Multi-symbol Angluin-style Mealy machine learning

Having described the motivations for developing DAE2FSM, we now present the core techniques underlying DAE2FSM.

As mentioned in Section 4.1, DAE2FSM aims to abstract a given SPICE netlist as an FSM, while capturing the analog non-idealities exhibited by the given circuit.

Here, we would like to first note that the FSM produced by DAE2FSM needs to be a Mealy machine. Recall from Section 3.2 that Mealy machines are FSMs that take in an input symbol sequence and produce an output symbol *at each FSM state transition*. In general, this output symbol will depend on both the current state of the system and the input symbol just read. Thus, for every input sequence, the FSM will return an output sequence of length equal to that of the input sequence. This is important because this mirrors how we expect typical circuits to behave. Given a continuous input waveform between times t_0 and t_f, we are usually interested in the response produced by the circuit in the time interval $[t_0, t_f]$. When the interval $[t_0, t_f]$ is discretized, this corresponds to a Mealy Machine like behaviour.

We point this out because the core techniques underlying DAE2FSM are inspired by Angluin's algorithm [43]. However, Angluin's algorithm in its standard form does *not* produce Mealy Machines. Instead, it produces a different kind of FSM called a Deterministic Finite Automaton (or DFA). DFAs are FSMs that take in an input sequence, and instead of producing an output symbol at every transition, they produce a *single output symbol* in response to the entire input sequence. This output symbol will either be a "1" (indicating that the FSM "accepts" the given input sequence) or a "0" (indicating that the FSM "rejects" the input sequence). This does not exactly capture the behaviour exhibited by the circuits we are interested in, because analog circuits behave much more like Mealy Machines than like DFAs. Also, the circuits we are interested in can typically produce a variety of output symbols, which are not limited to just 0 and 1. This again, is more Mealy Machine-esque than DFA-like. Therefore, in this section, we modify Angluin's algorithm to learn Mealy

machines rather than DFAs.

The algorithm itself can be thought of as a sequence of interactions between two entities: one of these is called the *Teacher*, and the other is called the *Learner*. The Teacher is assumed to know everything about the circuit being Booleanized. In practical terms, this means that the Teacher has access to the given SPICE netlist, as well as a SPICE simulator that is able to simulate the given netlist on a variety of inputs. The Learner, on the other hand, works entirely in the Boolean domain: it has no information about the SPICE netlist, and no access to the SPICE simulator. Indeed, the only way the Learner obtains information about the underlying circuit is by asking specific questions of the Teacher. This is similar to the concept of RESTful APIs[2] [50] in web design frameworks: the circuit is like the backend database, the Teacher is like a server (with access to the database) that implements a RESTful API, and the Learner is like a client that repeatedly queries the server API and pieces together the information returned by the server over many stateless transactions.

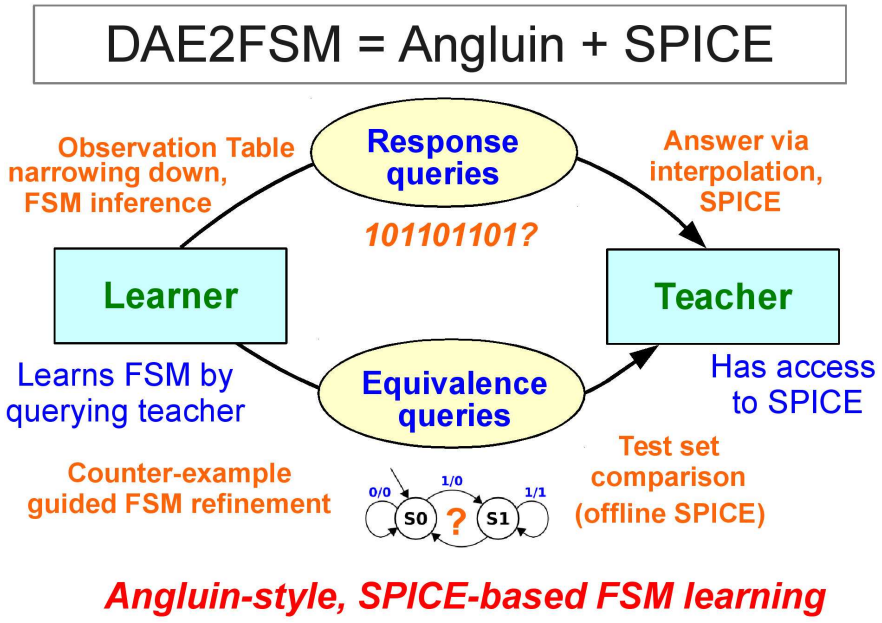


Figure 4.2: The high-level flow behind DAE2FSM: Learning Mealy Machines based on a sequence of interactions between a Teacher and a Learner.

Eventually, the Learner learns enough information about the circuit to construct an FSM abstraction for it. This abstraction is then repeatedly refined by the Learner (by making further queries of the Teacher). Finally, this process converges to an FSM, which is returned as the result produced by DAE2FSM. This is illustrated in Fig. 4.2.

Before the learning process begins, both the Teacher and the Learner decide on an input alphabet Σ, and an output alphabet Γ. These can be any finite sets of symbols, and in

[2]Representational State Transfer Application Programming Interfaces.

DAE2FSM, these are usually chosen based on the given circuit's inputs and outputs. For example, if the circuit has a single input that is discretized into 8 levels (or 3 bits), the input alphabet Σ will be a set of 8 symbols (*e.g.*, $\{a,\ b,\ c,\ d,\ e,\ f,\ g,\ h\}$), with each of these symbols mapping to a specific analog value for the circuit's input. Similarly, if the circuit has a single output that is discretized into 4 levels (or 2 bits), the output alphabet Γ will be a set of 4 symbols (*e.g.*, $\{w,\ x,\ y,\ z\}$), with each of these symbols mapping to a specific analog value for the circuit's output.

As mentioned above, the Learner obtains information about the circuit by asking specific questions, or *queries*, of the teacher. These queries can take two forms: (a) *I/O queries* or *response queries*, and (b) *equivalence queries* or *FSM checks*.

An *I/O query* (or *response query*) involves the Learner presenting the Teacher with an *input sequence* (any sequence of symbols from the input alphabet Σ), to which the Teacher responds with an *output sequence* (a sequence of symbols from the output alphabet Γ, of length equal to that of the input sequence).

Given an input sequence, the Teacher answers the response query as follows (Fig. 4.3). First, the Teacher constructs an input *waveform* from the input *sequence*. An example is shown in Fig. 4.3. In this example, the input alphabet is the set $\{0,\ 1\}$, and the input sequence is 100110. Recall that symbols from the input alphabet usually have a direct relationship to the underlying circuit's input signals. In the example of Fig. 4.3, this relationship is relatively straightforward: the symbol 0 means a particular input voltage (say, 0 Volts), and the symbol 1 means a different input voltage (say, 0.8 Volts). Thus, the input sequence 100110 translates to an input waveform that starts off at 0.8 Volts, switches to 0 Volts at the next clock cycle, remains at 0 Volts for another clock cycle, then switches back to 0.8 Volts, and so on Fig. 4.3. Given any such input sequence, the Teacher is thus able to produce a corresponding input waveform (or waveforms, if the circuit has multiple inputs).

Next, the Teacher SPICE-simulates the given circuit on the input waveforms produced above. This is straightforward: the Teacher simply appends a few lines describing the input waveforms at the end of the SPICE netlist provided for the circuit, and then runs the SPICE simulator (for example, asking the SPICE simulator to run a transient simulation). The SPICE simulator runs the requested simulation and returns the circuit's response (the output waveforms corresponding to the supplied input waveforms) back to the Teacher.

The Teacher then discretizes the output waveforms above into a sequence of symbols from the output alphabet Γ. This is usually done by sampling and quantization: the output waveforms are sampled at regular intervals of time and then quantized into the output alphabet, producing a sequence of output symbols of length equal to that of the input sequence supplied by the Learner. Finally, the Teacher returns this sequence of output symbols back to the Learner, as the "answer" to the Learner's response query.

Thus, with each response query, the Learner increases its knowledge about the circuit. Eventually, the Learner learns enough about the circuit to derive an FSM abstraction (in the form of a multi-symbol Mealy machine) for it. The Learner's methodology for deriving this abstraction is described further below in this section.

Once the Learner derives an FSM abstraction, the second type of query (the equivalence

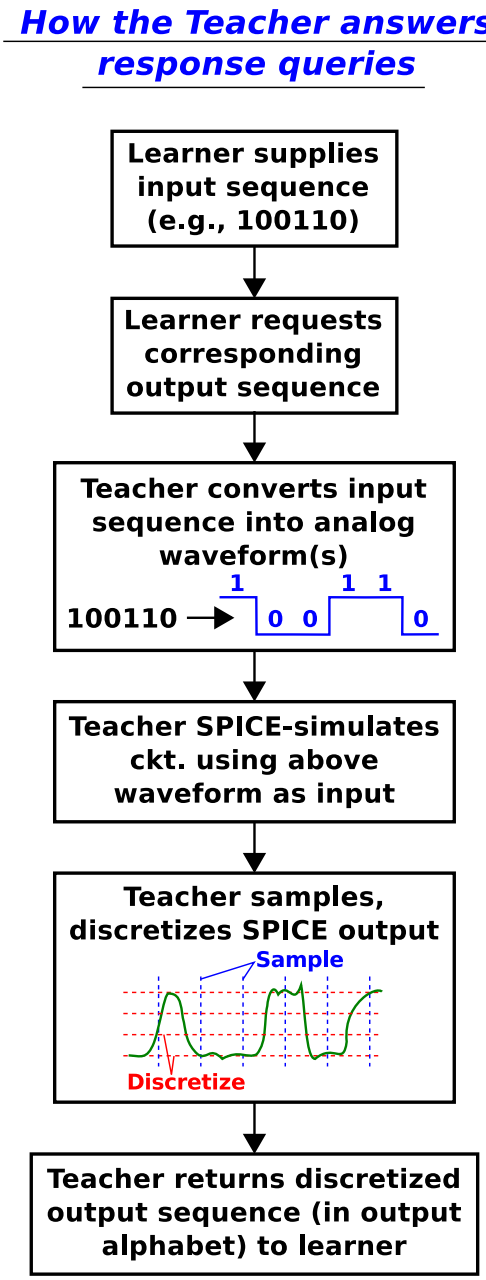


Figure 4.3: Flowchart showing how the Teacher answers a response query from the Learner.

query, or FSM check) comes into play. This query simply asks the Teacher whether the FSM produced by the Learner is a "good enough" approximation to the underlying circuit. If the Teacher answers "yes" to this query, the algorithm stops and the FSM that gained the Teacher's approval is returned by DAE2FSM. On the other hand, if the Teacher answers "no" to the equivalence query, the Teacher also supplies a *counter-example* showing why the FSM abstraction is not good enough to model the given circuit. This counter-example is then used by the Learner to refine the generated FSM (using a process described later in this section). This process of refinement continues until the Teacher answers "yes" to an equivalence query. This flow is illustrated in Fig. 4.5. The figure contains a number of additional details, including references to an observation table and such. These details are explained further below in this section.

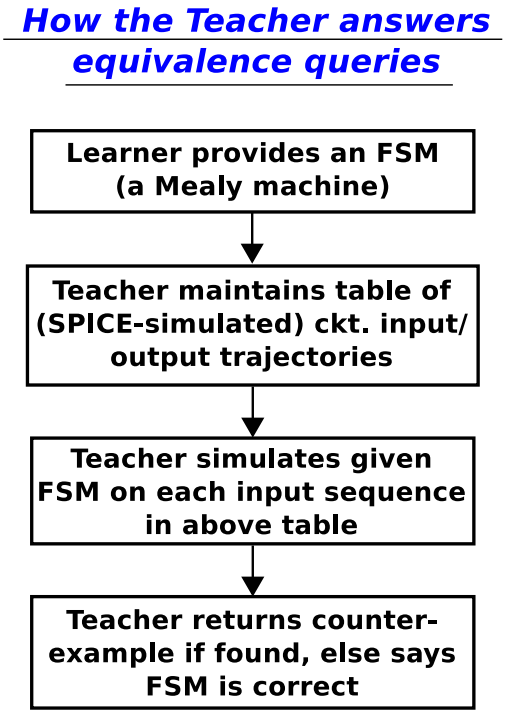


Figure 4.4: Flowchart showing how the Teacher answers an equivalence query from the Learner.

Given an FSM proposed by the Learner, the Teacher answers an equivalence query as follows. The Teacher runs a large number of SPICE simulations (which can be chosen either deterministically or randomly, or a combination of the two) of the given circuit, exercising the circuit with a wide variety of input sequences/waveforms, and recording the corresponding output sequences/waveforms (using the same discretization and interpolation procedures described above that the Teacher uses for answering response queries). This allows the Teacher to build a table of input sequences and their corresponding output sequences. The Teacher then uses this table to test the FSM proposed by the Learner. That is, for every

input sequence in this table, the Teacher simulates the FSM and checks whether the output sequence produced by the FSM is indeed identical to the one recorded in the table. If the Teacher finds a discrepancy between the FSM and SPICE, it flags the discrepancy as a counter-example and returns the "offending" input sequence to the Learner along with a "no" answer to the equivalence query, giving the Learner an opportunity to refine the FSM. On the other hand, if the Teacher is unable to find such a discrepancy, it returns a "yes" answer to the equivalence query, and the algorithm terminates.

Here, we would like to pause, and make a note that the Teacher has enormous freedom in the way it converts an input *sequence* into an input *waveform*: for example, as described above, the Teacher can interpret input symbols as *quantized voltage levels* (*e.g.*, symbol "a" could mean 0 Volts, symbol "b" could mean 0.1 Volts, *etc.*), leading to a *multi-level-discretized* learned FSM. Alternatively, the teacher can also interpret input symbols as *bit vectors* specifying Boolean values for multiple circuit inputs (*e.g.*, symbol "a" could mean the bit-vector 000, symbol "b" could mean 001, *etc.*); this interpretation results in a *multi-input* Mealy machine abstraction for the circuit. Yet another possibility is that the Teacher can interpret the input symbols as *switching events* (*e.g.*, symbol "a" could mean that the clock switches, symbol "b" could mean that the data switches, *etc.*), which results in a different kind of multi-input FSM that sometimes offers greater intuition into circuit dynamics (*e.g.*, see our FSMs for latches and flip-flops in Section 4.4 and Section 4.5). This *freedom to interpret the input alphabet in multiple ways* is an important aspect of DAE2FSM: it allows us to use the same fundamental framework (automated multi-symbol Mealy machine learning) to generate multi-level, or multi-input, or multi-output, or any combination of these, FSM abstractions, depending on the circuit-driven application at hand. For example, if the application is to characterise a failing flip-flop (see Section 4.6), a multi-level FSM could be the best option. On the other hand, if the application is a combinational/sequential circuit such as a counter, a multi-input, multi-output FSM might be the best suited (see Section 4.7).

Having presented the Teacher's side, we now discuss the Learner's algorithm (shown, along with an example, in Fig. 4.6). At any point, the Learner maintains two sets of *words* (a word is a sequence of symbols) over the input alphabet. These sets of words are denoted S and E. Initially, at the start of the algorithm, S and E are both initialized to $\Sigma \cup \{\epsilon\}$, where ϵ denotes the empty string.

In addition to S and E, the Learner maintains an *observation table* T that contains all the information acquired about the circuit (by querying the Teacher) thus far.

This observation table is organized as follows. Each *row* in the observation table corresponds to a word from the set $S \cup S.\Sigma$ where . denotes concatenation. This corresponds to all the words in S, plus all the words obtained by adding any character in Σ to any word in S. Each column in the observation table corresponds to a word from E. At any time, for every ordered pair $(s, e) \in (S \cup S.\Sigma) \times E$, the observation table T contains an entry $T(s, e)$, which is equal to the last $|e|$ output symbols returned by the Teacher for the input sequence $s.e$. To take a simple example, let $s = 111$ and $e = 0000$. Let us say the Learner has already conducted a response query for $s.e$, which would be 1110000. Let us say the Teacher has returned *cbbabcd* as the answer to this response query. Then, $T(s, e)$ (in this

case, $T(111, 0000)$) would hold the value *abcd*, the 4 output symbols corresponding to the e part of the $s.e$ response query. We note that, in Angluin's original algorithm for DFAs, T only contained 0 and 1 entries. However, to extend the algorithm to multi-symbol Mealy machine learning, we need to store output strings in T instead of just binary values.

For each $s \in S \cup S.\Sigma$, let row(s) denote the entire row corresponding to s in T.

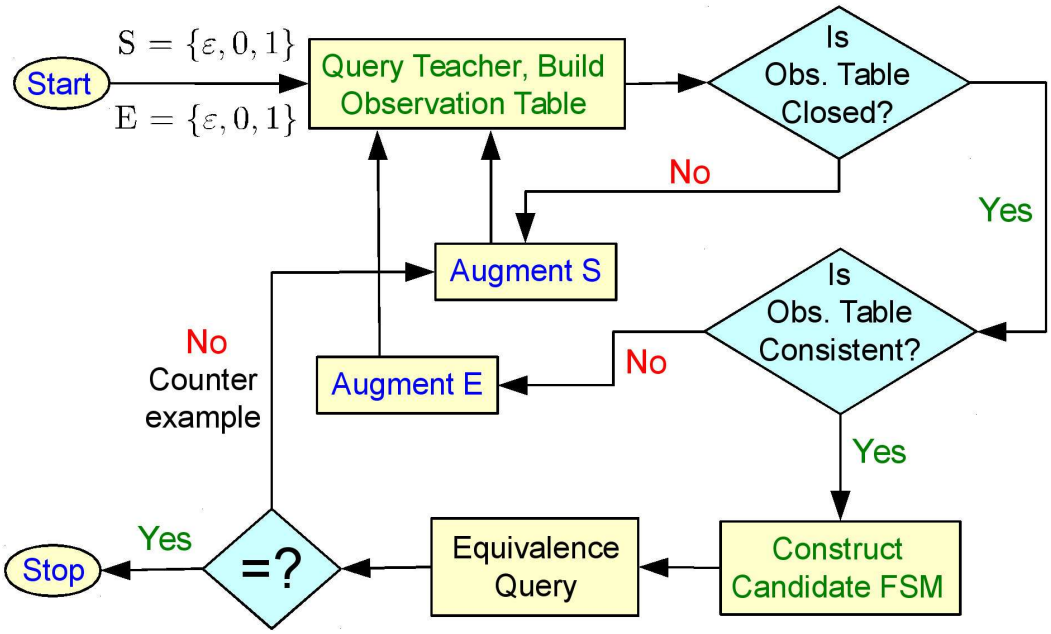


Figure 4.5: Flowchart showing the high-level mechanism followed by the Learner in DAE2FSM to produce an FSM abstraction of the given circuit.

Using the information in T, the Learner tries to match an *FSM state* to each word in S. That is, for each $s \in S$, the Learner tries to associate with s the *final* state reached by the FSM when fed the input sequence s. For this, T must satisfy two conditions:

Closedness: For each $s_1 \in S.\Sigma$, there must exist $s_2 \in S$ such that row(s_1) = row(s_2). The intuition is that: the set of FSM states associated with S is incomplete if there is no destination state for an input in $S.\Sigma$. If T is not closed, then the strings violating closedness must be added to S, and T repopulated (by conducting further response queries).

Consistency: For every $s_1, s_2 \in S$ such that row(s_1) = row(s_2), it should also be true that row($s_1.\sigma$) = row($s_2.\sigma$) for every $\sigma \in \Sigma$. The intuition is that: one cannot associate identical FSM states with two different input strings (s_1 and s_2) in S, unless one can also associate identical states with $s_1.\sigma$ and $s_2.\sigma$, for every $\sigma \in \Sigma$. If T is not consistent, then there must exist a string $e \in E$ such that $T(s_1.\sigma, e) \neq T(s_2.\sigma, e)$. We find such a string e, add $\sigma.e$ to E, and repopulate T (by conducting further response queries).

Now, we are finally in a position to describe the full algorithm followed by the Learner. The Learner starts with a small observation table T corresponding to $S = E = \Sigma \cup \{\epsilon\}$

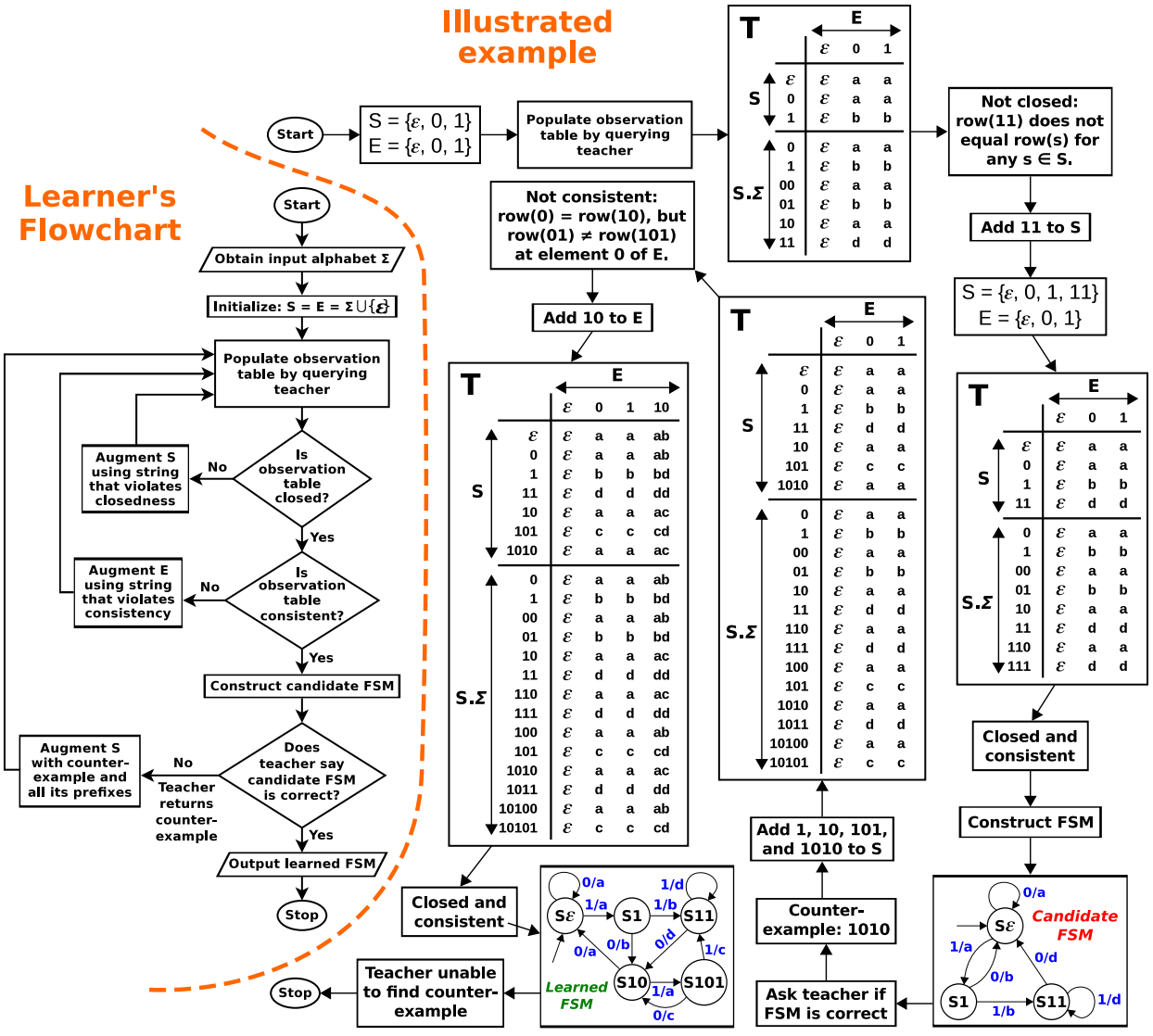


Figure 4.6: Flowchart with an accompanying example showing in detail the mechanism followed by the Learner in DAE2FSM to produce an FSM abstraction of a failing D-flip-flop circuit (for additional details, see Section 4.6).

(which is populated by issuing response queries to the Teacher). In general, this observation table is not guaranteed to be closed or consistent. So the Learner keeps issuing response queries to the Teacher as described above until T is both closed and consistent.

At this point, when T is finally both closed and consistent, the Learner produces an FSM (Fig. 4.5). The states in this FSM are associated with strings in S: the Learner simply takes all the *unique* rows in T corresponding to the strings in S, and each unique row becomes a state in the generated FSM. The arcs in the FSM are then populated based on the data in T, as follows. Let us consider the action taken by an FSM state that is associated with the word $s \in S$ (call this state A), when presented with the input symbol $\sigma \in \Sigma$. Clearly, if the input to the FSM is s, the FSM ends up in state A. The question is, if the input to the FSM is $s.\sigma$, what is the state that the FSM ends up in? This is answered by reading off row$(s.\sigma)$ from T, which, by the closedness property, will correspond to some row in the unique set of rows above. Let us say B is the FSM state corresponding to this row. The Learner then adds an arc from state A to state B in the FSM, with the input symbol σ and the output symbol given by $T(s, \sigma)$. By repeating this process for all states in the FSM and all symbols in the input alphabet, the Learner populates all the arcs in the FSM.

Once the FSM is ready, the Learner gives it to the Teacher and conducts an equivalence query (Fig. 4.5). If the Teacher finds a counter-example, the Learner adds this counter-example to S (Fig. 4.5). Then it repopulates T until it becomes closed and consistent, constructs a new FSM, and a new equivalence query results (Fig. 4.5). This is repeated until the Teacher is unable to find a counter-example to the Learner's FSM. In Fig. 4.6, we present the complete Learner's algorithm as a flowchart, and also illustrate it with an example. The example shows how the Learner, starting from scratch, learns a multi-output FSM for a failing D-flip-flop (for more details, see Section 4.6).

Having presented the core techniques underlying DAE2FSM, the next few sections present some results obtained by applying DAE2FSM to "digitalish" systems plagued by analog non-idealities. As mentioned earlier, we have applied the techniques discussed above to generate multi-symbol Mealy machine abstractions of latches, flip-flops, and circuits constructed from them. Below, we discuss these results in detail.

The next few sections are organized as follows: we begin by generating binary FSM abstractions of correctly functioning latches and flip-flops (Section 4.3), for different timing relationships between the clock (`CLK`) and data (`D`) signals. We then use *multi-symbol* FSM learning to construct *multi-input* FSMs for latches and flip-flops (Section 4.4, Section 4.5); this encodes all relevant switching patterns of `CLK` and `D` using a multi-symbol alphabet (*i.e.*, the algorithm automatically generates a single FSM capturing the circuit's behaviour across all relevant timing scenarios). After this, in Section 4.6, we apply *multi-level* discretization to realize FSM abstractions of error-prone flip-flops, which fail on certain inputs because of analog effects. Finally, in Section 4.7, we present a *multi-input, multi-output, combinational/sequential* application: that of automatically learning a state machine abstraction for a 0-to-5 increment/decrement counter implemented with 280 transistors.

4.3 Example: Binary FSMs for correctly functioning latches and flip-flops

We start with a simple case: generating binary FSMs for latches and flip-flops. We consider six different timing scenarios for the clock and data signals, and we demonstrate that DAE2FSM is able to produce Mealy machines whose predictions match well with SPICE-level simulations in all six scenarios.

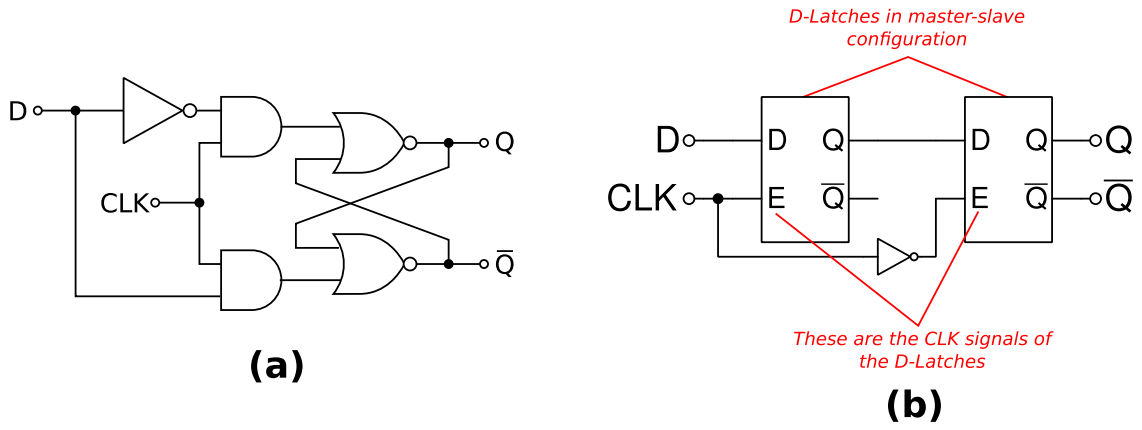


Figure 4.7: The D-Latch (left, part (a)) and the D-flip-flop (right, part (b)) circuits that we Booleanized using DAE2FSM. Figure credits: Parts of this figure were copied from Wikipedia and then modified.

Fig. 4.7 shows the D-Latch and the D-flip-flop circuits that we Booleanized using DAE2FSM. We modelled these circuits as continuous dynamical systems in full SPICE-level detail, using BSIM4 device models for every transistor in these circuits. We then Booleanized these SPICE netlists using DAE2FSM, as described below. Note that both the D-Latch and the D-flip-flop, in the continuous domain, are two-input, two-output systems (with the inputs being the clock `CLK` and the data `D` signals, and the outputs being the stored value `Q` and its complement $\overline{\texttt{Q}}$).

For now, let us limit the target FSMs to a single input (*i.e.*, the data input `D`). The other input (the clock `CLK`) is fixed, and assumed to be a periodic pulse, whose one period is shown in Fig. 4.8 (right). It is assumed that `D` transitions exactly once per clock cycle. Also, the input/output sequences required for DAE2FSM are sampled exactly once per clock cycle (either before, or after `D` transitions).

This gives rise to six possible timing scenarios, tabulated in Fig. 4.8. For each scenario, we used DAE2FSM to learn FSMs for both a D-latch and a D-flip-flop.[3] Fig. 4.8 shows that, depending on the scenario and the circuit, the learned FSM can be either a *buffer* or a *delay*.

[3]We note that a D-latch FSM has also been learned by the authors of [1]. However, that work considers only one of the scenarios outlined in Fig. 4.8, whereas we analyse all possible scenarios. Also, the authors of [1] consider only latches, whereas we have developed FSMs for latches, flip-flops and beyond.

Input changes	Input, output are sampled	Scenario #	Latch FSM	Flip-Flop FSM
Only when CLK is low	When CLK is low (before input changes)	1	Buffer	Buffer
	When CLK is low (after input changes)	2	Delay	Delay
	When CLK is high	3	Buffer	Delay
Only when CLK is high	When CLK is low	4	Buffer	Buffer
	When CLK is high (before input changes)	5	Buffer	Buffer
	When CLK is high (after input changes)	6	Buffer	Delay

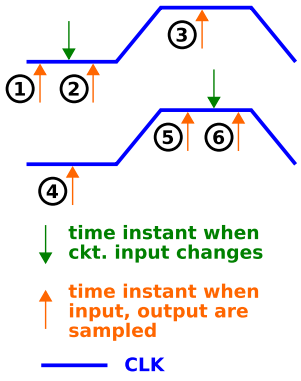


Figure 4.8: Binary FSMs learned for latches and flip-flops in various operating modes.

These FSMs are shown in Figs. 4.9 (a) and 4.9 (b) respectively; it is seen that the buffer FSM simply relays its input directly to the output (*i.e.*, its input and output sequences are always identical), whereas the delay FSM shifts the input to the right by one element. Together, the two FSMs capture the behaviour of ideal latches and ideal flip-flops in their various operating modes (Fig. 4.8).

For example, we know that an ideal D-latch can operate in two modes: it is transparent when `CLK` is high, but retains its output (even if the input changes) when `CLK` is low. Thus, if the input transitions (once a clock cycle) when `CLK` is low, and input/output sequences are sampled just before `CLK` turns high (scenario 2 in the above table), the output would reflect the previous input (applied 1 clock cycle earlier), which corresponds exactly to a delay FSM (as indeed, the table above shows).

Similarly, we know that an ideal D-flip-flop captures the value of `D` precisely at the negative edge of each clock cycle (*i.e.*, during the interval when `CLK` transitions from high to low), and remains opaque to changes in `D` at all other times. Fig. 4.9 (c) shows a SPICE simulation[4] of such a flip-flop's output (the blue waveform) where the input `D` (the green waveform) transitions, once per clock cycle, when `CLK` (the black waveform) is low.

From Fig. 4.8, we see that a buffer FSM predicts the flip-flop's output at uniformly spaced time points just *after* the negative edge of the clock (*i.e.*, scenario 1). For example, given the input sequence of Fig. 4.9 (c), the buffer FSM predicts the output sequence 0100110101 at these time points. This digital output sequence can now be mapped back to an *analog* output sequence, by tagging each output symbol with a specific analog voltage (determined at the time of sampling I/O sequences for the FSM learning algorithm). For instance, in this example, we tag the output symbol "0" with the analog voltage 0 Volts, and the output

[4]All SPICE simulations in this chapter have been carried out with 22nm devices, using BSIM4 device models. We obtained device parameters from [51], and the SPICE engine from [52].

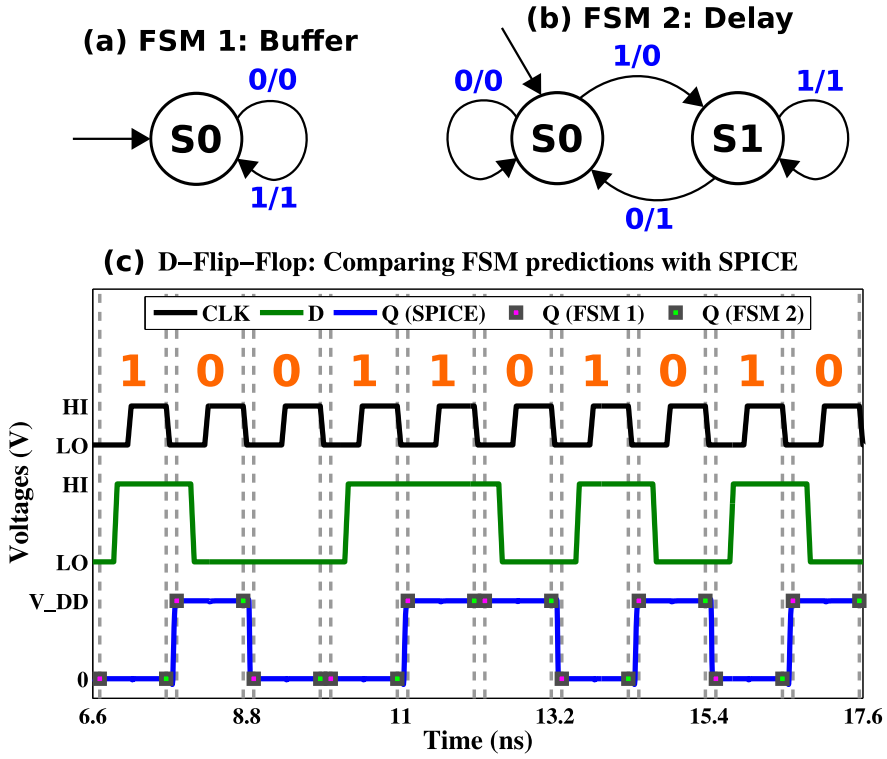


Figure 4.9: The *buffer* and *delay* FSMs returned by DAE2FSM accurately reflect the behaviour of an ideal D-flip-flop.

symbol "1" with the voltage V_{DD} Volts (which, in this case, is 0.8 Volts). Hence we obtain a sequence of analog voltages (in this case, [0V, 0.8V, 0V, 0V, 0.8V, 0.8V, 0V, 0.8V, 0V, 0.8V]) associated with uniformly spaced time points. We now overlay this analog time series on top of the SPICE waveform (see the magenta markers on top of the blue SPICE waveform in Fig. 4.9 (c)), to judge how well the discrete-domain FSM is able to predict the circuit's continuous-domain behaviour. In this case, just from a visual examination, it is clear that the FSM's predictions do in fact, tally closely with the SPICE-simulated waveform. Similarly, we have also plotted (with green markers in Fig. 4.9) the analog time series version of the delay FSM's predictions (which we know, from Fig. 4.8, to be valid just *before* the negative edge of the clock). From the figure, we see that this set of predictions also closely matches the SPICE waveform at these time points (the grey vertical lines).

4.4 Example: Multi-input FSMs for correctly functioning latches

The FSMs of the previous section handle only one circuit input (*i.e.*, D); the other input (CLK) is taken to be a fixed waveform that is known *a priori*. This necessitates many separate learnings, one for every possible switching pattern of D relative to CLK (as tabulated in Fig. 4.8). Moreover, if the input can transition twice or more per clock cycle, none of the FSMs learned above would be valid, and a fresh set of FSMs would need to be learned. The learned FSMs must then be "pieced together" to understand the functionality of the given circuit. This process can be tedious and inconvenient.

Instead, DAE2FSM offers the capability of learning *multi-input FSMs* (as outlined in Section 4.2), thereby dispensing with the need to generate/piece together many one-input FSMs. By considering D and CLK as two separate circuit inputs, the algorithm automatically learns a multi-input FSM that fully takes into account all relevant switching combinations of both inputs. We now demonstrate the multi-input capability of DAE2FSM on a D-latch.

To produce the multi-input latch FSMs (explained in Section 4.2 and in Section 4.1), we first encode all relevant switching patterns of both inputs, using a 4-symbol input alphabet $\Sigma = \{w, x, y, z\}$ for the multi-symbol Angluin procedure outlined in Section 4.2. Input symbol w indicates that both CLK and D are held constant (until the next sampling instant). Similarly, input symbol x (y) indicates that only CLK (D) switches (becomes high if it was low earlier, and vice-vera), while D (CLK) is held constant. Finally, symbol z indicates that both CLK and D switch their values: z therefore has two different meanings, depending on whether CLK switches first or D switches first. The two meanings lead to two different multi-input FSMs, as we show below.[5] Clearly, this 4-symbol input alphabet can represent all possible sequences of (legal) switching events in both inputs. For example, if one wants to determine the latch output following three switches of D before a single switch of CLK, and then a clock switch, and then two more switches of D, one simply passes the input sequence $[y, y, y, x, y, y]$ to the Mealy machine learned by DAE2FSM.

Fig. 4.10 (b) shows the multi-symbol Mealy machine automatically learned by DAE2FSM for the D-latch, for the case that CLK switches ahead of D on input symbol z. This FSM has two *transparent* states (TR1 and TR2), and four *opaque* states (OP1 to OP4). These states offer intuition into how the latch functions: every switch in CLK (*i.e.*, input symbol x or z) causes the FSM state to toggle between transparent and opaque. By contrast, a switch in D does not affect the transparency or opacity of the current FSM state. Closer examination reveals that when CLK goes low (*i.e.*, the latch transitions from transparent to opaque), the FSM always reaches the opaque state with the correct polarity (*i.e.*, OP1 if D is low at the instant the clock switches, OP4 otherwise). Transitions from opaque to transparent states also reflect precisely how one would expect an ideal latch to behave.

[5] Alternatively, one could also learn a 5-symbol FSM, where the two meanings of z are encoded by two different input symbols z_1 and z_2; however, the 4-symbol approach has the added advantage that the resultant FSMs can be combined via non-deterministic transitions to model race conditions in the input (Fig. 4.10 (f)).

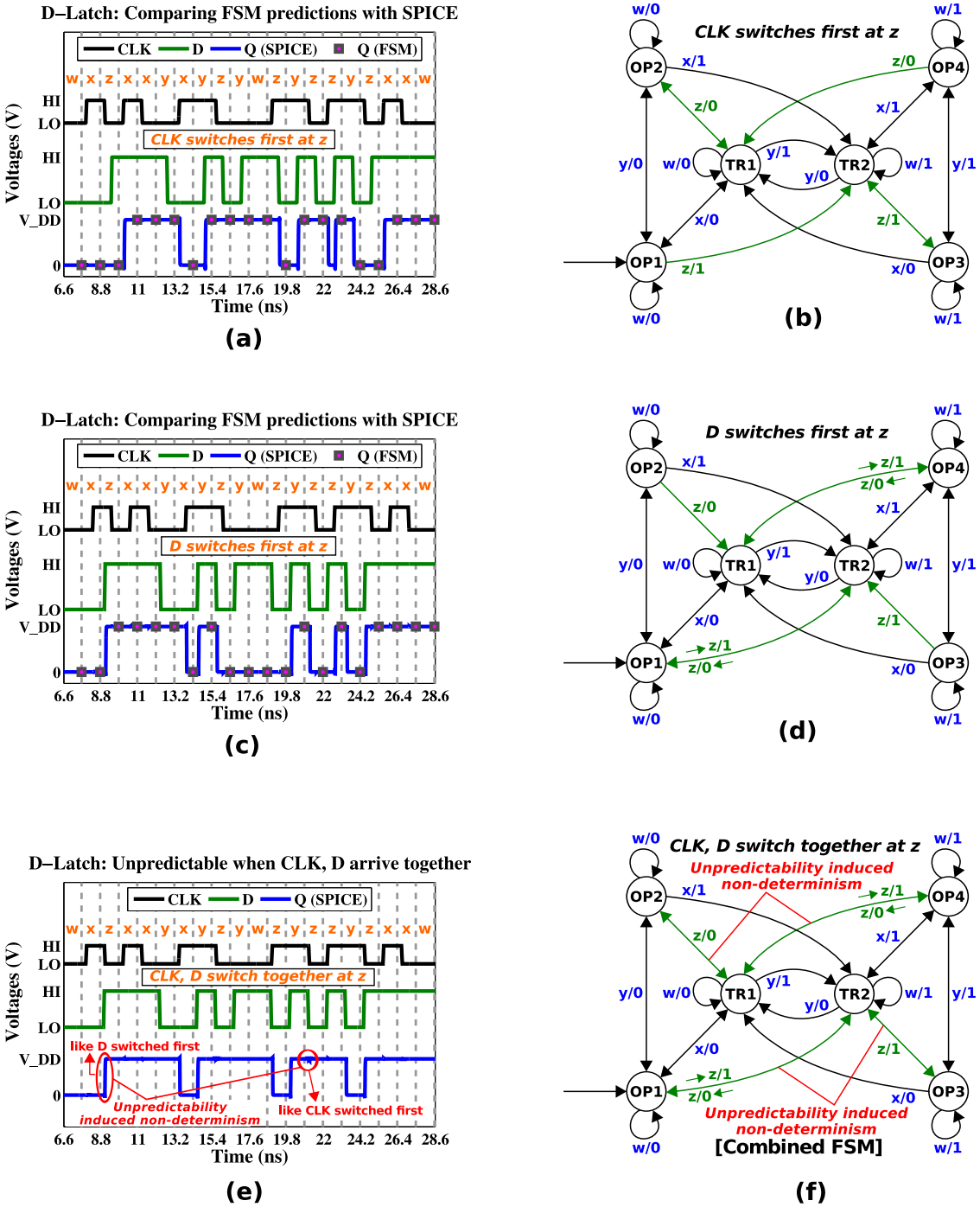


Figure 4.10: Multi-input DAE2FSM applied to construct multi-input FSM abstractions of a D-Latch.

Indeed, as Fig. 4.10 (a) shows, the latch output (Q) predictions made by this FSM closely tally with SPICE simulations, even for a complicated sequence of input switches that cannot be handled by any binary-only FSM derived earlier.

Fig. 4.10 (d) depicts the FSM derived when D switches ahead of CLK at input symbol z (with the meaning of the other input symbols unchanged). As expected, the FSMs in Figs. 4.10 (b) and 4.10 (d) are identical except for transitions on input z: their states are in one to one correspondence for all z-less input sequences. This FSM's predictions are also in excellent agreement with SPICE simulations (as seen from Fig. 4.10 (d)).

Thus, our multi-input DAE2FSM technique has automatically produced Mealy machines that accurately mimic the latch's behaviour under all relevant switching conditions. The only caveat is that the algorithm does not (yet) automatically handle illegal race conditions in the input;[6] for example, if CLK switches to low, and D switches at exactly the same time (a well-known "illegal" situation that can produce unpredictability, and even metastability), the output of a D-latch can become unpredictable, which the learned FSM, being a deterministic automaton, fails to capture. This unpredictability is illustrated in Fig. 4.10 (e): when both CLK and D switch simultaneously, the latch sometimes behaves as though CLK switched first, and sometimes as though D switched first. To account for such conditions, we combine the FSMs in Figs. 4.10 (b) and 4.10 (d) to arrive at a non-deterministic FSM. The rationale is that CLK in Fig. 4.10 (b) switches ahead of D at input symbol z, while Fig. 4.10 (d) applies when D switches ahead of CLK at input symbol z. Therefore, if CLK and D switch at the same time, the latch could (in theory) choose to behave according to either of these FSMs; in practice, the latch's "choice" of FSM would depend on many factors, including the exact shapes of the switching input waveforms, clock jitter, voltages at internal nodes, device parameters, noise processes (*e.g.*, thermal noise, shot noise), *etc.*. Since most of these factors are inherently unpredictable, it is convenient to abstract them by introducing non-deterministic transitions in the learned FSM. This results in the Mealy machine of Fig. 4.10 (f), whose non-z transitions are identical to the original FSMs, but whose z-transitions include non-determinism.

4.5 Example: Multi-input FSMs for correctly functioning flip-flops

We now repeat the analysis of the previous section, but for a D-flip-flop[7] instead of a D-latch. The results (Fig. 4.11) roughly mirror those of the previous section (Fig. 4.10); however, there are interesting differences, as noted below.

As with the D-latch, we have generated FSM abstractions of the D-flip-flop using the multi-symbol input alphabet $\{w, x, y, z\}$ (where the symbols have the same meaning as

[6] We are currently working on improvements to DAE2FSM that can handle race conditions, metastability, *etc.*.

[7] The flip-flop we have used is a master-slave, negative-edge triggered D-flip-flop built from two D-latches.

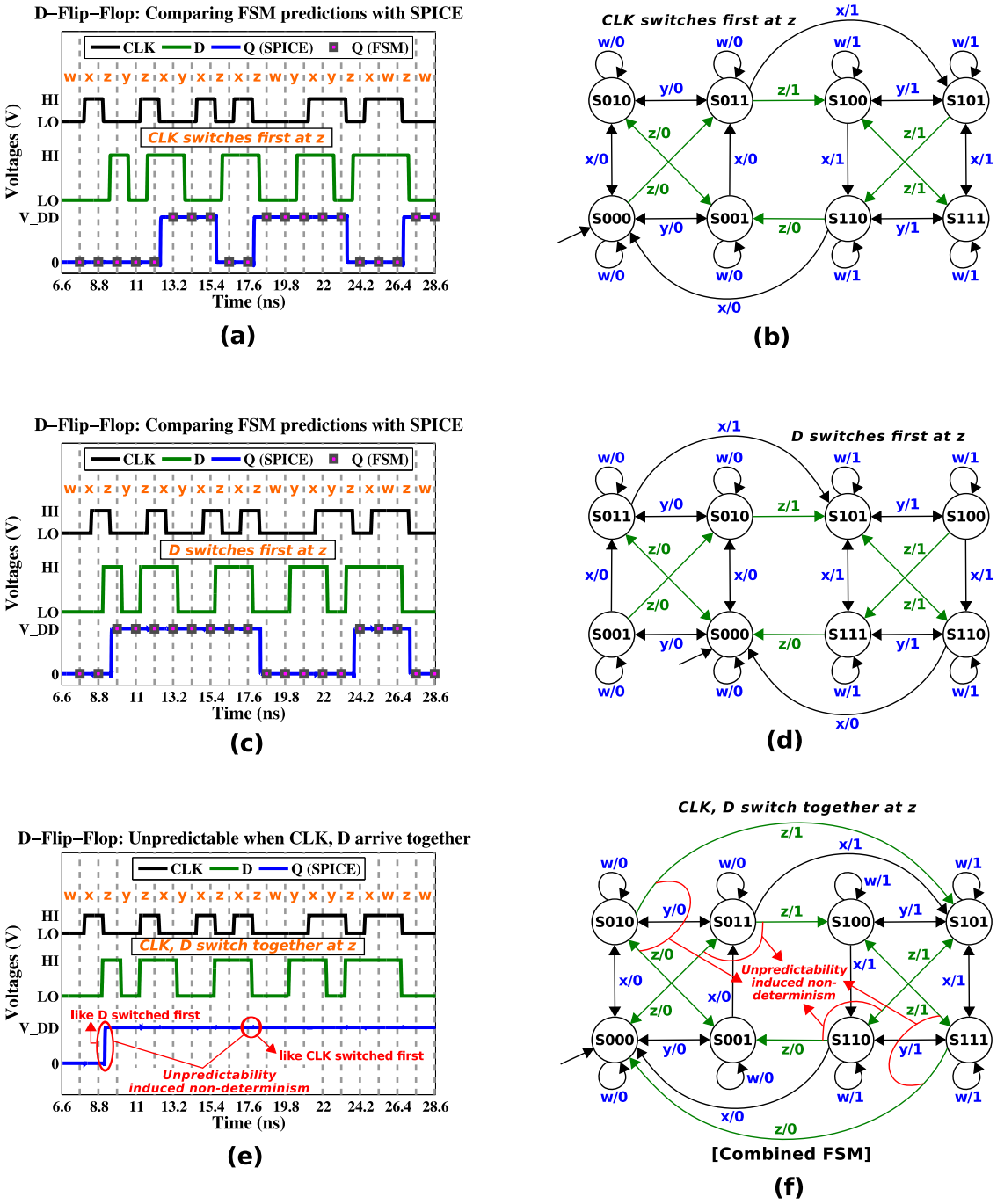


Figure 4.11: Multi-symbol DAE2FSM applied to construct unified FSM abstractions of a D-flip-flop.

before). The auto-generated FSMs are shown in Figs. 4.11 (b) and 4.11 (d).

Unlike the D-latch, however, there is no concept of a "transparent" or "opaque" state in the D-flip-flop's FSMs. Rather, the intuition is that each state can be viewed as an ordered 3-tuple, whose dimensions are the stored flip-flop value `Q`, the clock `CLK`, and the input `D`. For example, state `S101` indicates that: (a) the flip-flop currently stores a value `Q` = 1 (captured at the most recent negative clock edge), (b) the clock is low, and (c) `D` is high. With this intuition, it is readily seen that the FSMs in Figs. 4.11 (b) and 4.11 (d) capture the precise functionality expected of a D-flip-flop: the input is captured and relayed to the output exactly once per "cycle" of the clock, and this happens only when `CLK` transitions from high to low. Moreover, as shown in Figs. 4.11 (a) and 4.11 (c), the predictions made by these FSMs match very well with values sampled from SPICE simulations.

Finally, Fig. 4.11 (e) shows that the flip-flop's output can be unpredictable when `CLK` switches from high to low, and `D` also switches at exactly the same time. In such a situation, the flip-flop behaves at times as if `CLK` switched first, and at other times as if `D` switched first. Moreover, as before, we observe that the two FSMs learned for the different meanings of z, in this case also, behave identically for all z-less sequences, with their states being in one-to-one correspondence. Therefore, as before, it is possible to combine these FSMs by introducing non-deterministic transitions. The resulting combined FSM is shown in Fig. 4.11 (f).

4.6 Example: Multi-level FSMs for failing flip-flops

Having shown how DAE2FSM produces correct abstractions of properly functioning latches and flip-flops, we now demonstrate another crucial feature of DAE2FSM: that it can abstract useful FSMs even for circuits that suffer from such significant analog imperfections that digital functionality is compromised. The motivation is that it is often important to characterise the behaviour of latches and flip-flops functioning under non-ideal operating conditions (*e.g.*, under lowered supply voltages, extreme overclocking, a particularly unfavourable process variability corner, *etc.*). Such characterisation, for example, plays a central rôle in the design of error-resilient communication systems. We note that the generation of (binary) FSMs for non-ideal latches has already been demonstrated in [1]. In this work, we focus on *multi-level discretizations* applied to flip-flops and demonstrate how DAE2FSM captures novel failure modes.

Fig. 4.12 shows two possible failure modes (discovered by DAE2FSM) that a D-flip flop can exhibit: both result from overclocking the flip-flop (*i.e.*, as the clock frequency increases, the flip-flop has less time to capture the input value at the negative clock edge, eventually being unable to do so for some input sequences). One failure mode (Fig. 4.12 (a)) can be adequately captured by a binary FSM, whereas the other (Fig. 4.12 (c)) needs a multi-level output alphabet (supported only by the multi-symbol approach).

In the first mode of failure[8] (Fig. 4.12 (a)), a single "1" applied at the input of the flip-flop

[8]This failure is observed at a clock frequency of 10.42 GHz, a reasonable frequency at which to expect the flip-flop to fail on its own, *i.e.*, without the added effect of combinational delays from external sources.

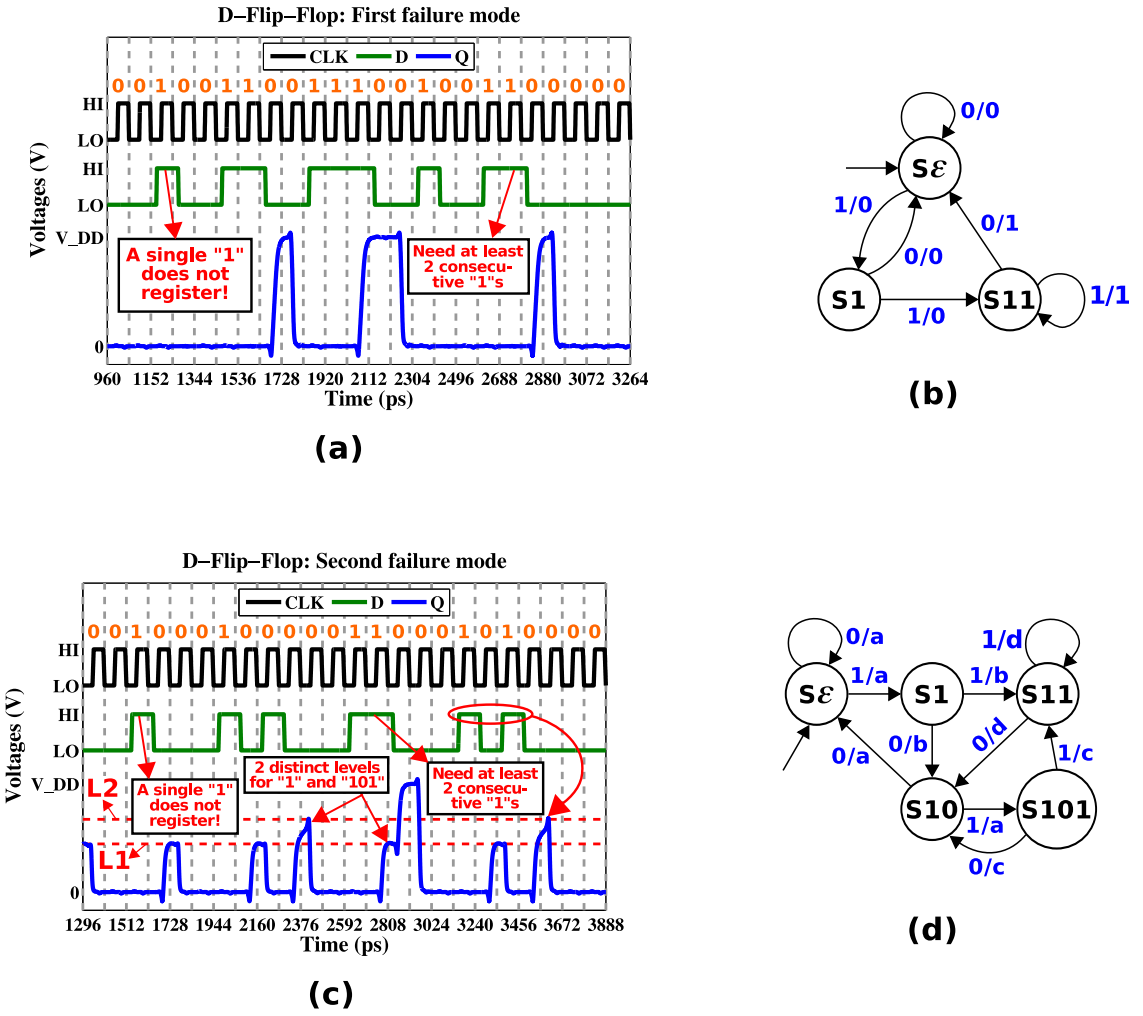


Figure 4.12: Failure modes observed for flip-flops, and FSMs learned for failing flip-flops.

fails to register at the output; the flip-flop's response is too slow to capture the fleeting bit. However, if the applied "1" remains in place for two or more consecutive clock cycles, the flip-flop is able to register it, because its internal (analog) state has already been nudged in the right direction by the first "1". This is illustrated by the SPICE plots in Fig. 4.12 (a). As shown in Fig. 4.12 (b), DAE2FSM is able to capture this failure mode. The FSM in Fig. 4.12 (b) starts at the "zero" state marked $\mathtt{S}\epsilon$. In this state, if the FSM encounters a "1", it moves to an *intermediate state* `S1`, where it waits for the next input. If the next input is "0", the FSM goes back to its initial state without registering the previously applied "1" (reflecting the failing flip-flop's behaviour). If the next input is also "1", the FSM enters the "one" state (and stays there for as long as the input remains "1") because it has witnessed at least two consecutive "1s" at the input. Thus, the binary FSM of Fig. 4.12 (b) is adequate to capture this failure mode.

The second failure mode, illustrated in Fig. 4.12 (c), occurs when the flip-flop is clocked at 9.26 GHz. The output of the flip-flop clearly shows 4 distinct levels: at 0, V_{DD}, and two intermediate levels indicated by the horizontal, dashed red lines (marked L1 and L2). These intermediate levels appear in the output only when specific sequences are detected at the input. For example, level L1 appears when the input sequence contains a "1" that is preceded by neither a "1" nor the sequence "10". On the other hand, level L2 is reached whenever the input sequence "101" is applied at the input. This can be explained by recognizing that the failing flip-flop has a memory longer than one clock cycle: each time a "1" is applied, the failing flip-flop remembers it for the next two clock cycles, which has an effect on its output during that time-span. However, because the output is now a 4-level signal, it cannot be reproduced by a binary FSM. In this case, therefore, one needs an FSM with an output alphabet of at least 4 symbols, one for each output level (the input alphabet can still be binary). Applying multi-level DAE2FSM to this circuit results in the multi-symbol Mealy machine of Fig. 4.12 (d). The example of Fig. 4.6 shows in detail how DAE2FSM was able to learn this Mealy machine from scratch. It can be verified that this FSM produces output "c" (corresponding to level L2) on input sequence "101", and output "b" (level L1) on a "1" that is preceded by neither a "1" nor the sequence "10". The output is "a" (the symbol for 0 Volts) for a "0" at the input, and "d" (the symbol for V_{DD} Volts) only if the input has two or more consecutive "1s". Thus, multi-level DAE2FSM can be applied to find state machines that model failing flip-flops.

4.7 Example: FSM abstraction of a 280-transistor BSIM4 counter circuit

Having developed FSM abstractions of basic units such as latches and flip-flops, we now apply DAE2FSM to a much larger design: a 280-transistor, multi-input, multi-output circuit that includes both combinational and sequential logic elements. Although the circuit is considerably more complex than the examples above, the learned FSM reproduces its behaviours perfectly.

The circuit is a 0-to-5 increment/decrement counter with reset (schematic shown in Fig. 4.13), which takes two (digital) inputs (not counting `CLK`) X and R, and returns three (digital) outputs $\mathtt{Q}_1$, $\mathtt{Q}_2$ and $\mathtt{Q}_3$. The output bits $\mathtt{Q}_1$ to $\mathtt{Q}_3$ encode a whole number (the count) in the range 0-to-5 (0 and 5 included), with $\mathtt{Q}_1$ ($\mathtt{Q}_3$) being the most (least) significant bit. At the negative edge of each clock cycle, the count is either incremented (where 5 "increments" to 0), decremented (where 0 "decrements" to 5) or reset to 0, depending on the inputs supplied. The "reset to 0" action (denoted R) is taken whenever the reset input (also R) is set; otherwise, the count is incremented (denoted $+$) if $X = 1$, and decremented (denoted $-$) if $X = 0$. Fig. 4.14 (a) shows the state transition table associated with the counter, while Fig. 4.14 (b) shows SPICE simulations of a complete increment cycle and a complete decrement cycle of the counter (with a reset in between). These simulations confirm that

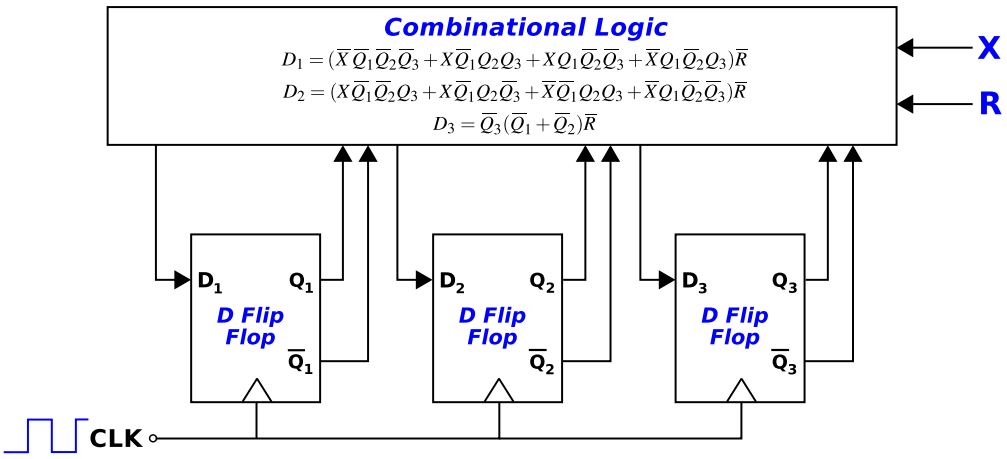


Figure 4.13: Schematic of a 0-to-5 increment/decrement counter with reset (please see Fig. 4.14 (a) for the corresponding state transition table).

the counter functions exactly as intended.

We applied DAE2FSM to learn a Mealy machine representation of this counter automatically. Each of the two inputs and three outputs was discretized using two levels; they were then encoded using 4 and 6 symbols,[9] respectively, for multi-symbol Angluin-based learning (see Section 4.2).

Fig. 4.14 (c) depicts the Mealy machine learned by DAE2FSM; it consists of six states `S0` to `S5` (arranged in a circle in the figure), corresponding to the count values 0-to-5. Increment actions, taken when $(X, R) = (1, 0)$, result in clockwise traversal of the circle, while decrement actions, taken when $(X, R) = (0, 0)$, result in anti-clockwise traversal. At each state, the reset action (at $R = 1$) results in a state transition back to `S0`. This state machine captures the intended logical functionality of the counter exactly. Also, the output sequence predicted by the learned Mealy FSM, when translated to analog values (as described earlier), matches SPICE-level simulations well (see the magenta markers in Fig. 4.14 (b)).

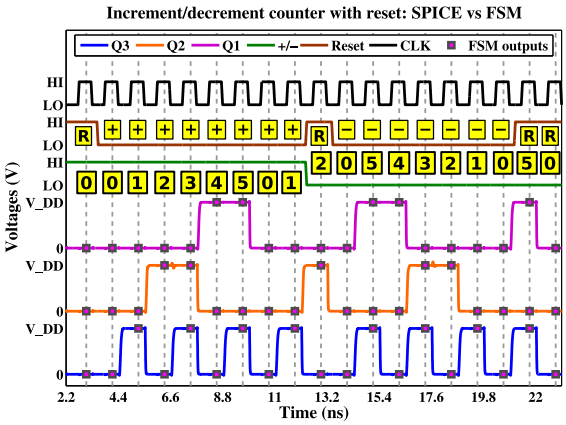
4.8 Limitations of DAE2FSM

While DAE2FSM works well for Booleanizing "digitalish" designs (such as those in the examples above), it suffers from several limitations that make it ineffective when applied to "genuinely analog" designs, *i.e.*, those that exhibit (and are designed to exhibit) bona fide analog dynamics, as opposed to small analog non-idealities on top of predominantly digital dynamics. Below, we describe some of these limitations, and how they inspired us to develop non-black-box Booleanization techniques like ABCD-L and ABCD-NL (the topics of the next two chapters).

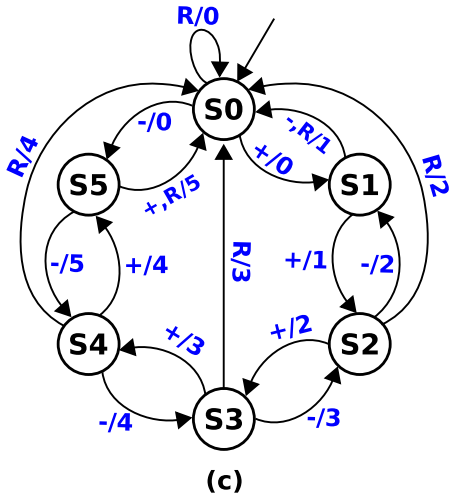
[9] Ordinarily, 8 symbols are needed to encode three digital outputs; however, two of these (bit vectors 110 and 111) never appear in this counter's output.

Inputs			Current state	Next state
X	R	Action	$Q_1\ Q_2\ Q_3$	$D_1\ D_2\ D_3$
0/1	1	R	Don't care	0 0 0
1	0	+	0 0 0	0 0 1
			0 0 1	0 1 0
			0 1 0	0 1 1
			0 1 1	1 0 0
			1 0 0	1 0 1
			1 0 1	0 0 0
0	0	−	0 0 0	1 0 1
			0 0 1	0 0 0
			0 1 0	0 0 1
			0 1 1	0 1 0
			1 0 0	0 1 1
			1 0 1	1 0 0

(a)



(b)



(c)

Figure 4.14: (a) Table showing the state transitions of a 0-to-5 increment/decrement counter with reset (please see Fig. 4.13 for a circuit schematic). The counter takes the increment (decrement) action + (−) when $X = 1$ ($X = 0$), unless the reset bit R is set, in which case the counter is reset to 0. (b) SPICE simulation showing a complete increment cycle and a complete decrement cycle of the counter. The top row of yellow boxes indicate the next "action" that will be taken by the counter, while the bottom row indicates the current count. (c) Multi-symbol Mealy machine automatically learned by DAE2FSM for the counter.

- **Oracles are difficult to implement.** Recall, from Section 4.2, the Teacher/Learner framework that DAE2FSM uses to Booleanize a given system. In this framework, there are 2 requirements that the Teacher must satisfy: it should be able to answer response queries, and it should be able to answer equivalence queries. The first requirement (answering response queries) is, of course, easy enough for a Teacher that has access to a SPICE simulator. The second requirement, however, requires the Teacher to essentially be an *oracle*. Given a candidate FSM, the onus is on the Teacher to determine whether the FSM represents the underlying circuit well (and, if not, provide a concrete counterexample). This is a very difficult problem. In our Teacher implementations, we usually solve it by having the Teacher simply simulate the learned FSM on a reasonably large number of input sequences, and compare the corresponding output sequences against a pre-computed database of correct input/output sequence pairs. A simple tweak of this approach also tests the learned FSM on some randomly generated input sequences, in addition to the pre-determined sequences above. However, such approaches are by no means exhaustive; they can easily miss important corner cases, and therefore the learned FSM can fail to capture design bugs. Moreover, since the equivalence checking routine of the Teacher often needs to be written by the AMS designer, it is now the designer's responsibility to make sure that all corner cases are tested. As a result, many non-ideal system behaviours that the designer is unaware of, or incorrectly considers unimportant, may not be tested for, and may thus not be captured by the generated Boolean model. In some ways, therefore, writing this equivalence checker suffers from the same pitfalls that manual Booleanization does (Section 3.9). Also, for most practical real-world AMS designs, the number of corner cases is simply too large to list, so the equivalence checking process may need to be significantly overhauled (*e.g.*, by including a bounded model checking step, or a sequential equivalence checking step, within the Teacher's equivalence checking procedure). This can make answering equivalence queries very costly. For many AMS designs, therefore, the need for an oracle can be a severe limitation.

- **DAE2FSM scales poorly with circuit "memory".** Many genuinely analog systems have "long memories". That is, an input to the system that is applied now could significantly affect the system's output a long time from now. A simple example for this is an interconnect with large parasitics; a pulse that is applied at the input can take a long time (essentially a "dead delay") to arrive at the output. To take another example, let us consider a simple communication channel (frequently modelled as an RC or RLGC chain). It is well-known that the bits that are transmitted earlier through such channels can significantly interfere with bits that are transmitted much later; this phenomenon is called inter-symbol interference (ISI). It is not uncommon for real-world channels to feature ISI at lengths 30 unit-intervals or more, *i.e.*, bit i could appreciably interfere with bit $i + 30$. These effects are very common in genuinely analog systems, but much less so in the kinds of "digitalish" systems we have seen in this chapter.

Let us imagine how DAE2FSM would behave when applied to such a system. Because such systems have long memories, the counterexamples returned by the Teacher are likely to be long. In the communication channel above, for example, the Teacher could return a counterexample of length 30. Long counterexamples are particularly problematic for DAE2FSM. This is because, every time a counterexample is returned, the Learner needs to add the counterexample, *as well as all its prefixes*, *as well as all words obtainable by appending a single input symbol to either the counterexample or any of its prefixes*, to the rows of the observation table. Thus, for a counterexample of length 30, it is conceivable that 60 rows would be added to the observation table. If the observation table has, say 20 columns, the Learner would need to initiate 1200 response queries, corresponding to 1200 SPICE simulations that the Teacher needs to carry out. This is just the first round; the responses to these queries, more often than not, result in an observation table that is either not closed or not consistent. This, in turn, spawns further response queries, and those spawn still further response queries, and so on.[10] In this way, it is not uncommon for a *single* longish counterexample to result in 10000 SPICE simulations. Worse, systems with long memories tend to result in *many* long counterexamples. For instance, in the communication channel example above, there are 2^{30} possible input bit sequences of length 30. If even a miniscule fraction of these were to turn into counterexamples, the Teacher would be forced to run several billion SPICE simulations. In our experience, when DAE2FSM is applied to such systems, the number of response queries required quickly spirals out of control. For example, we experienced this first-hand when we tried applying DAE2FSM to simple analog systems like RC filters/chains.

- **DAE2FSM scales poorly with input/output alphabet sizes.** This is another key limitation of DAE2FSM. In "digitalish" systems, most signals tend to be either 0 or 1, which is why many FSMs in this chapter feature small input and output alphabets. However, while Booleanizing systems with genuinely analog signals, it is necessary to discretize these signals more finely. For example, we may wish to represent each analog signal using 3 bits, or 8 levels of discretization. As the sizes of the input and output alphabets grow, the observation tables constructed by DAE2FSM tend to grow much faster. This is because, with more input symbols, the Learner needs to issue more response queries. For example, let us say that the Learner would like to know the system's responses to all possible input sequences of length 4 (this is artificially small; typically, the Learner will require many more responses to complete the observation table). If the input alphabet is just $\{0, 1\}$, then the number of response queries requested would be $2^4 = 16$. But if we were to discetize the circuit's input using 8 levels of discretization, this number would grow dramatically, to $8^4 = 4096$. And if the circuit had multiple input signals, each discretized using 8 levels, this number

[10] This is not dissimilar to the Greek myth chronicling the struggles of Heracles against the many-headed serpentine monster Hydra: each time Heracles cut off one of Hydra's heads, two grew back.

would grow still more quickly. In typical DAE2FSM runs, we believe that the Learner would end up issuing an exponential number of response queries.

- **DAE2FSM scales poorly with the Boolean model's clock period.** "Digitalish" systems usually have a well-defined clock, and one can usually use the same clock for the Boolean model as well. However, analog systems do not have clocks. While Booleanizing such systems, in order to capture the shapes of the underlying analog waveforms with good accuracy, it is necessary to choose a fine clock (*i.e.*, with a small clock period) for the Boolean model. Unfortunately, as the clock period of the Boolean model becomes smaller, the number of response queries required by DAE2FSM grows exponentially. For example, let us consider the Booleanization of a simple RC filter (shown at the top left of Fig. 5.4), using just 2 levels to discretize its input (corresponding to either 0V or 1V), but 8 levels to discretize its output. Clearly, we would like to capture the progression of the output waveform through 8 levels between 0V and 1V, so the Boolean model's clock period should be at most $1/8^{th}$ of the filter's time-constant. Also, clearly, this system has a "memory" of at least 8 Boolean clock cycles (indeed, by any practical definition, the "memory" of this circuit would be much larger).[11] Therefore, just to capture the response of the system to simple step waveforms, DAE2FSM would end up requesting about $2^8 = 256$ response queries (and probably many more). And as described above, these response queries would spawn further response queries. Thus, DAE2FSM's poor scaling with the clock period used for the Boolean model may force the designer to use larger clock periods, which can adversely impact the accuracy that can be obtained while Booleanizing genuinely analog systems with DAE2FSM.

- **DAE2FSM is not guaranteed to terminate.** When Angluin first published her technique for DFA-learning, she was able to prove that, as long as the Teacher had a DFA in mind, the Learner would be able to learn the DFA (and minimize it as well) [43]. However, in the case of DAE2FSM, the Teacher does *not* have a DFA in mind. The Teacher has a SPICE-level circuit (*i.e.*, a DAE) in mind. This DAE features a continuous state space, and no two states in this state space are quite identical. The Learner, on the other hand, knows nothing about this continuous state space. Anytime the Learner sees behaviour that is even a little bit different from previously seen behaviour, the Learner creates new FSM states. For example, let us again consider an RC filter (shown at the top left of Fig. 5.4). Consider two (discrete) sequences of inputs that have been applied to the RC filter, one of which has resulted in the current voltage across the capacitor being 0.56V, and the other has resulted in 0.57V. If one had a white-box technique (as opposed to DAE2FSM, which is a black-box technique)

[11] This is another key problem with DAE2FSM. As the Boolean model's clock period is made smaller (to ensure better fidelity to analog wave shapes), the memory of the system, in terms of the number of "clock cycles", increases. And, as we know, DAE2FSM does not work well for systems with long memories in relation to the clock period of the Boolean model.

for Booleanizing this circuit, one might construct an FSM that ends up in the same discrete state for both these input sequences. This may be reasonable, because we may not expect the Boolean model to distinguish between such "close" continuous states. However, the Learner in DAE2FSM does not see things this way. If a single input sequence exists (either in the Teacher's answers to the Learner's response queries, or in a counterexample returned by the Teacher to one of the Learner's equivalence queries), that elicits even a slightly different response from these two states (*e.g.*, a small timing difference that, when discretized, causes a shift of 1 Boolean clock cycle in the response of one of these states relative to the other), the Learner would have to view these as two completely different states. Therefore, depending on the precise implementation of the Teacher (in particular, the logic used by the Teacher for equivalence checking), the Learner might be trying to learn an infinite state machine, and not a finite state machine. So the algorithm might never terminate. Of course, when running this algorithm on a finite precision computer, we will be saved by the fact that SPICE itself is equivalent to Boolean model simulation (Section 3.5), and therefore the program will terminate. But the learned FSM can still be very large, and DAE2FSM can still take a very long time to learn it.

- **DAE2FSM is not guaranteed to produce FSMs without artifacts.** We know that the FSMs learned by DAE2FSM do not (and cannot) capture the continuous-time behaviour of the underlying circuit perfectly. But we do expect at least a qualitative match between such FSMs and SPICE (as described in Section 3.4). However, if the equivalence checking logic used by the Teacher is too lenient (and this is entirely possible, from our discussion above on how hard it is to write the equivalence checker), the learned FSM could have artifacts like "fake fixed points" (described in Section 6.3.2). Such artifacts can cause the FSM to make egregiously wrong predictions that do not even qualitatively match SPICE. Given a large FSM learned by DAE2FSM, it is also not easy to tell whether it contains such artifacts. For small FSMs (such as the ones in this chapter), the absence of such artifacts can usually be concluded just from visual inspection, but this does not hold true for FSMs with more than a few dozen states. This can severely limit the applicability of DAE2FSM. This issue is a consequence of DAE2FSM being a black-box technique with no preconceptions about the FSM being learned. The techniques developed in the next couple of chapters, ABCD-L and ABCD-NL, on the other hand, are white-box techniques. They guarantee, by construction, that the FSMs they produce will not contain such artifacts as "fake fixed points".

- **DAE2FSM does not take advantage of continuity, linearity, *etc.*.** This is again a consequence of the Learner not having the notion of "analog". To the Learner, all operations are done on symbols. Whether two symbols are "close together" or "far apart" (when mapped to the continuous domain), the Learner treats them the same way – as completely different entities. The same holds true for FSM states that are "close together" or "far apart". Because the Learner does not have these notions built-

in, it cannot take advantage of the fact that it is Booleanizing an AMS system. In contrast, the white-box Booleanization techniques developed in the next two chapters *do* take advantage of concepts like linearity and continuity. For example, ABCD-L (Chapter 5) uses a very special Boolean model structure (consisting of independent logic units whose outputs are combined together by a domain transformation unit) that is optimized to take advantage of the dynamics of LTI systems. ABCD-NL (Chapter 6) also has a special Boolean model structure (consisting of DC and TRAN FSM states) to capture non-linear dynamics in an intuitive way; in addition, it has a variety of "jump heuristics" that take advantage of the continuity of the AMS system being Booleanized. DAE2FSM does not have these advantages.

- **DAE2FSM can only produce FSMs in state transition graph (STG) form.** It is well-known that FSMs in STG form can grow large, consume a lot of memory, and be difficult to formally analyze/verify. In contrast, FSMs encoded in Boolean ciruit form (*e.g.*, as sequential AIGs) can represent the same dynamics much more compactly, and are frequently also easier to formally verify. Unfortunately, at this time, DAE2FSM can only learn STGs. We believe that extending it to learn FSMs in Boolean circuit form is a non-trivial research problem that requires rethinking the entire observation-table based flow of the Learner (and perhaps the API between the Learner and the Teacher as well).

- **DAE2FSM does not support overapproximations or underapproximations.** DAE2FSM, at this time, only learns deterministic FSMs that approximate circuit behaviour. But, in the verification community, it is considered good practice to use non-deterministic FSMs that overapproximate or underapproximate circuit behaviour, in order to model circuit dynamics conservatively, and hence add a margin of safety to the verification flow. Unfortunately, DAE2FSM does not support this. We believe that extending the core techniques underlying DAE2FSM, so as to produce overapproximate or underapproximate Boolean models, is a non-trivial research problem.

4.9 Summary

To summarise, in this chapter, we have developed and demonstrated DAE2FSM, a technique for automatically learning FSM abstractions of "digitalish" systems; these are systems whose end-to-end functionality is intended to be purely digital, but which exhibit significant analog non-idealities that often compromise correctness and performance.

To accurately capture the analog details and waveforms underlying these systems, we model them as fully continuous-domain dynamical systems, using full SPICE-level netlists and advanced device models such as BSIM4. We then Booleanize these netlists using DAE2FSM, producing FSMs that are able to reproduce the analog dynamics of these systems.

Our core technical contribution in this chapter is that we have extended the well-known Angluin's algorithm from computational learning theory, and adapted it to carry out multi-symbol Mealy machine learning (as opposed to simple 0/1 DFA learning). Furthermore, we have developed a new methodology allowing us to adapt this algorithm to truly continuous dynamical systems. This is made possible because we have adapted the "Teacher" in Angluin's algorithm to interact with a SPICE simulator and use it as a tool while answering the Learner's queries. We have also been able to combine multiple deterministic FSMs produced by this approach into a single non-deterministic FSM to capture *uncertain* circuit behaviour, allowing us to model features like metastability in a simple way.

The techniques above enable the generation of different classes of FSMs (including multi-level, multi-input, multi-output, and any combination of these) supporting different circuit-level applications. We have applied these techniques to automatically produce multi-input FSMs for properly functioning latches and flip-flops, by encoding all relevant input switching patterns with a single 4-symbol input alphabet. The auto-generated FSMs are able to produce output sequences that match well with SPICE simulations. We have also used the multi-symbol framework to learn multi-level FSM abstractions of error-prone flip-flops, where DAE2FSM was able to identify two failure modes in overclocked D-flip-flops. We have also generated a multi-input, multi-output Mealy machine abstraction of a 0-to-5 increment/decrement counter, which illustrates the applicability of our technique to a larger and more complex system than an individual latch/flip-flop.

However, DAE2FSM also has several limitations, many of which stem from the fact that it is a black-box technique where the Learner cannot take advantage of the fact that it is Booleanizing an underlying AMS system. In the next two chapters, we address some of these limitations by developing two new, non-black-box, Booleanization techniques: ABCD-L (Chapter 5), and ABCD-NL (Chapter 6).

Chapter 5

ABCD-L: Automated Booleanization of LTI Systems

This chapter presents ABCD-L, the second of three automated Booleanization techniques we have developed as part of the ABCD suite.

ABCD-L is meant for Booleanizing Linear Time Invariant (LTI) systems and components. These play an important rôle in many AMS designs today. For example, LTI systems include a wide variety of commonly used AMS building blocks, including amplifiers, communication channels, filters, on-chip interconnect networks, on-chip power grid networks, linearized circuits used for small-signal analysis *etc.*. Therefore, being able to Booleanize such components effectively is a valuable capability from the standpoint of accurate AMS modelling, high-speed simulation, formal verification, *etc.*. This is the capability that ABCD-L provides, and this chapter is devoted to discussing how ABCD-L works and how the generated Boolean models achieve a good balance between accuracy and scalability when applied to LTI components typically found in modern AMS designs.

Unlike the "digitalish" systems that were Booleanized in the previous chapter, the LTI systems considered in this chapter are genuinely analog in nature, *i.e.*, their intended functionality is itself analog, and is described in terms of continuous dynamical systems such as systems of differential-algebraic equations. Booleanizing such genuinely analog systems presents a greater challenge than Booleanizing "digitalish" systems. This is because, typically, "digitalish" systems only exhibit a few analog kinks on top of their predominantly digital behaviour, whereas analog systems (such as the LTI systems we consider in this chapter) feature long memories, a high degree of inter-symbol interference and dispersion, *etc.*, rendering Booleanization techniques like DAE2FSM ineffective (as described in Section 4.8).

In this chapter, we model analog LTI systems as linear DAEs that take the following form:

$$C\frac{d}{dt}\vec{x} + G\vec{x} + B\vec{u} = \vec{0}, \text{ and}$$
$$\vec{y} = c^T\vec{x}, \tag{5.1}$$

where $\vec{x}$ denotes the state of the LTI system, $\vec{u}$ denotes its inputs, and $\vec{y}$ denotes its outputs. The symbols C, G, B and c are constant valued matrices. All quantities are continuous.

We then carry out Booleanization of the DAE above, taking advantage of the rich literature and mathematical tools and techniques that have been developed for LTI systems. For example, a key step in our Booleanization involves eigen-analysis, which takes advantage of the fact that the system being Booleanized is LTI. Where necessary, we also take advantage of the rich theory of LTI model order reduction to produce Boolean models that scale well with respect to system size. Another nice feature of ABCD-L is that one can provably generate Boolean models of *arbitrarily high accuracy*, just by using more bits to discretize the underlying signals and by using a faster clock for the Boolean model.

As with other Booleanization algorithms presented in this dissertation, the Boolean models generated by ABCD-L can be used in conjunction with existing techniques for Boolean synthesis, formal verification, high-speed logic simulation, *etc.*. Furthermore, these Boolean models can also be used within hybrid system frameworks, enabling the user to accurately represent LTI dynamics without incurring the penalty of adding continuous variables that significantly slow down hybrid system verification tools.

We apply ABCD-L to I/O links composed of RC/RLGC units, capturing important analog effects like inter-symbol interference, overshoot/undershoot, ringing, *etc.* – all using purely Boolean models. We also present a continuous-time differential equalizer example, where ABCD-L accurately reproduces key design-relevant AMS metrics, including the eye diagram correction achieved by the circuit. Furthermore, for real-world LTI systems that are large (tens of thousands of nodes), we demonstrate that ABCD-L can be applied in conjunction with Model Order Reduction (MOR) techniques; we use this to produce accurate Boolean models of an industry-scale power grid network (with 25849 nodes) made available by IBM. We also demonstrate that Boolean simulation using ABCD-L's models offers considerable speed-up over standard circuit simulation using linear multi-step numerical methods.

For a more succinct treatment of the material in this chapter, we refer the reader to our paper on ABCD-L [53].

5.1 The need for ABCD-L

In today's advanced process technologies (32nm and below), LTI components in AMS designs (*e.g.*, interconnect/parasitic networks, I/O and equalization circuitry, wireline and wireless communication channels, *etc.*) are becoming key bottlenecks that impact system-level performance [3, 5]. Moreover, an increasingly significant proportion of overall design

bugs are now attributable to on-chip AMS components (please see the discussion in Section 1.2.2), and a significant fraction of these bugs have roots in sub-systems that exhibit LTI dynamics. For example, a recent internal study at Intel concluded that AMS modules account for over 20% of all design bugs in cutting edge microprocessors. A key source of such bugs lies in I/O, *i.e.*, the high-speed communication channels between CPU and DRAM, which are subject to inter-symbol interference, crosstalk, dispersion, *etc.*, by virtue of their LTI characteristics. Furthermore, such bugs tend to be difficult and costly to identify and correct, typically requiring extensive time-consuming SPICE-level simulations.

For early detection and timely correction of the above AMS-related design bugs, it is desirable to carry out functional validation and formal verification of AMS components *at or near SPICE-level accuracy*. However, in most existing approaches to AMS verification (please see Chapter 2 for an overview), the underlying model that is verified is usually a highly simplified abstraction that does not attempt to capture any of the SPICE-level subtleties (*e.g.*, layout-dependent parasitics, cross-talk, inter-symbol interference, ringing) that are responsible for bringing about design bugs/loss of performance – even in situations where such dynamics is predominantly LTI. Thus, while the simplified models currently in use by AMS verification tools can be useful for gaining intuition about the circuit's operation as a whole (as intended by the designer), they are of limited use when it comes to *debugging* AMS designs or *issuing performance guarantees*, because many parasitic performance-limiting LTI effects are not accounted for in the hybrid system models that are verified.

Recall (from Chapters 1 and 2) the distinction between the two kinds of formal verification techniques: *Boolean techniques* versus *"hybrid system" techniques*. Boolean techniques (*e.g.*, [6]) represent each underlying circuit signal as a discrete (usually binary) quantity, whereas hybrid system techniques (*e.g.*, [14, 25, 29–31, 34, 49]) offer the capability to represent signals as either discrete or continuous-valued quantities. However, the ability to represent and reason about continuous variables often comes at an enormous computational cost, which renders hybrid system techniques typically orders of magnitude slower than their Boolean counterparts. Indeed, while Boolean techniques are routinely used in the industry to verify circuits with millions of logic gates, even state-of-the-art hybrid system techniques are unable to verify systems with more than a few (*e.g.*, 5 to 10) continuous variables.

The *limited scalability* of hybrid system techniques is the main reason why AMS circuits are typically not modelled/verified at SPICE-level accuracy; instead, existing hybrid system methodologies are forced to adopt over-simplified "behavioural" AMS component models that often do not bear close resemblance to SPICE. This drastically limits their applicability in the context of AMS debugging/performance verification: without SPICE-accurate modelling, the predictions made by hybrid system based AMS verification engines are not reliable enough for designers. As a result, the prevailing practice amongst AMS designers today is to carry out time-consuming SPICE simulations rather than place their trust in AMS verification tools. To overcome such "designer skepticism", we believe that it is necessary to significantly scale up existing hybrid system techniques, so that they embrace SPICE-accurate models even for large AMS designs.

In this chapter, we propose a technique (called ABCD-L)[1] to bridge the gap between SPICE-level detail and the models used by AMS verification engines (both Boolean and hybrid), for an important subclass of AMS circuits, namely, Linear Time Invariant (LTI) systems. LTI systems constitute a fundamental class of AMS systems, including, for example, on-chip and off-chip interconnect, clock tree networks, amplifiers and filters, linearized small-signal equivalent circuits, channel models, many kinds of equalizers and I/O circuitry, *etc.*. The key idea behind ABCD-L is to approximate the continuous-time, continuous-valued dynamics of LTI systems using purely Boolean/discrete models. That is, given a set of differential equations for an LTI system (or alternatively, measured data, transfer function characteristics, scattering parameters, reduced order models, *etc.*), ABCD-L is a "push-button" style technique that produces as output a Boolean circuit abstraction (comprised entirely of Boolean logic elements such as registers, counters, *etc.*), that compactly encodes the analog behaviour of the given system, and is able to reproduce it while maintaining a good balance between accuracy and scalability (please see Section 3.7 for a detailed discussion of the tradeoff between modelling accuracy and scalability in the context of Booleanization). Another important feature of ABCD-L is that it can approximate LTI systems to *any desired level of accuracy* (see Section 5.2).

As with other Booleanization approaches in this dissertation, ABCD-L also works by first *discretizing* each underlying circuit signal, as well as the circuit's inputs and outputs. Please see Section 3.1 for a fuller discussion of the typical steps involved in Booleanizing a design, starting with discretization. As with DAE2FSM, the number of bits to be used to discretize these circuit signals, as well as the clock period to be used for this discretization, can be specified by the user. These are important levers for the user to explore the tradeoff between the accuracy and the size of the resulting Boolean model.

The discretized circuit signals are stored in Boolean *registers*. Given the contents of each register at a particular time instant, ABCD-L uses Boolean logic operations to approximate the *next time instant* at which these contents must be updated (*e.g.*, in response to changing input). For example, a Boolean *counter* can be used to keep track of time (at a sufficiently fine resolution, as described above), and there can be some *control logic* (also Boolean) to ensure that each register's contents are updated at the appropriate time. Thus, ABCD-L *transforms* the underlying LTI system into something like an *event-based discrete formulation*, realizable as a Boolean circuit. More details about the inner workings of ABCD-L can be found in Section 5.2. Also, please see Section 3.3 for a discussion on the broad flow involved in storing discretized circuit signals in registers and updating them using combinational logic as part of the Boolean model.

The approach above offers several compelling features. Firstly, because ABCD-L produces purely Boolean models, it is well-suited for use in conjunction with existing techniques for Boolean synthesis, verification, high speed logic simulation, *etc.*. By reducing LTI dynamics to Boolean form, ABCD-L makes it possible to leverage powerful Boolean techniques

[1] **A**ccurate **B**ooleanization of **C**ontinuous **D**ynamics - **L**inear.

for model checking/reachability analysis of LTI systems,[2] as well as Boolean systems coupled with LTI dynamics (*e.g.*, high-speed digital logic with parasitic interconnect). Secondly, in the context of AMS modelling/verification, we know that existing hybrid system approaches are unable to cope with more than a few continuous variables, whereas they can comfortably handle thousands of purely Boolean variables. Therefore, by enabling LTI dynamics to be accurately represented using "cheap" Boolean variables, ABCD-L frees up "precious" continuous variables for other purposes (*e.g.*, to accurately model non-linear dynamics). Therefore, with ABCD-L, it may be possible to expand the scope of hybrid system approaches to much larger systems than they can handle at present. Although this notion has been theoretically studied before (*e.g.*, see [42]), we believe that ABCD-L constitutes the first practical approach for Booleanizing continuous LTI dynamics in an accurate, systematic, and scalable manner.

Furthermore, ABCD-L also offers significant scalability advantages over explicit state-enumeration techniques like DAE2FSM [45], which produce Finite State Machines (FSMs) in State Transition Graph (STG) form, requiring exponential time and space complexity with respect to LTI system size. By contrast, ABCD-L's implicit, *circuit-encoded* models can be exponentially more compact than STGs. As a result, ABCD-L scales practically linearly with respect to both system size and the desired fidelity of the Boolean approximation, without blowing up in either runtime or model size. In addition, the above event-based Boolean formulation lends itself to a new methodology for accurate, high-speed LTI system simulation (covered in Section 5.2). This methodology, as we show in Section 5.7, can offer considerable speed-up over traditional circuit simulation methods based on linear multi-step integration [41]. Moreover, ABCD-L has been designed to handle stiff systems efficiently, and it can take full advantage of LTI Model Order Reduction (MOR) techniques to Booleanize large LTI systems in a scalable manner.

We have applied ABCD-L to representative LTI circuits such as I/O links composed of RC and RLGC units. For these systems, we show that the Boolean models produced by ABCD-L are able to accurately reproduce the original system's continuous dynamics, including important performance-limiting analog effects such as inter-symbol interference, overshoot/undershoot, ringing, *etc.*. In addition, we have applied ABCD-L to produce purely Boolean, small-signal models of a more complex, non-linear AMS system: an LTI channel followed by a differential equalizer (linearized using small-signal device models). In this case, too, ABCD-L is able to accurately capture and reproduce the circuit's dynamics in SPICE-level detail, using purely Boolean models all along. Further, ABCD-L is also able to capture higher-level design relevant AMS metrics, such as the eye diagram correction achieved by the equalizer. In addition, to accurately Booleanize industry-scale real-world LTI systems in a computationally viable manner, ABCD-L can be applied in conjunction

[2] In this chapter, we confine ourselves to the question of how to construct purely Boolean models that accurately capture analog LTI dynamics. *Formal verification* involving such models is an important next step, and one that is the subject of ongoing research. In this chapter, however, we do not seek to address the verification problem.

with LTI MOR techniques; we demonstrate this for a 25849-node benchmark power grid network made available by IBM (please see Section 5.6).

5.2 ABCD-L's core: Eigen-analysis and Booleanization of LTI systems

In this section, we describe the key ideas and core techniques underlying ABCD-L.

Fig. 5.1 depicts the ABCD-L flow, which takes as input an LTI system, and produces as output a purely Boolean approximation for it.

The given LTI system is assumed to take the following form:

$$C\frac{d}{dt}\vec{x} + G\vec{x} + B\vec{u} = \vec{0}, \text{ and}$$
$$\vec{y} = c^T\vec{x}. \tag{5.2}$$

In the above equations, $\vec{x}$ denotes the state of the LTI system, $\vec{u}$ denotes its inputs, and $\vec{y}$ denotes its outputs. The symbols C, G, B and c are constant valued matrices. All quantities are continuous.

However, for simplicity and ease of exposition, we will assume below that the LTI system is an ODE that takes the following form:

$$\frac{d}{dt}\vec{x} = A\vec{x} + B\vec{u}, \text{ and}$$
$$\vec{y} = c^T\vec{x}. \tag{5.3}$$

Here, $\vec{x}$ is the system's (continuous) analog state (a vector of voltages and currents), A is a real square matrix, $\vec{u}$ is the system's (time-varying) input, and $\vec{y}$ its corresponding output. We note that, if the ODE is not directly available, but instead only measured data (*e.g.*, from AC excitation at several frequencies), or S-parameters, or black-box transfer function characteristics, are available, ABCD-L can still obtain the requisite ODE by applying standard fitting techniques (*e.g.*, VectorFit [54], table-based methods [55], *etc.*).

We note that the techniques underlying ABCD-L are indeed capable of Booleanizing the more general DAE form of the equations above (Eq. (5.2)). However, this introduces special corner cases (such as purely algebraic, instead of differential, relationships between the elements of $\vec{z}$ and $\vec{u}$) that will unnecessarily complicate the explanations below. Therefore, in the discussion below, we restrict ourselves to LTI systems in ODE form (Eq. (5.3)).

As indicated in Fig. 5.1, ABCD-L begins with an eigen-analysis [56] of the above ODE system. Let us assume that the matrix A has an eigenvector $\vec{p}$, with eigenvalue λ [56]. Then, by definition, we have the following:

$$A\vec{p} = \lambda\vec{p}. \tag{5.4}$$

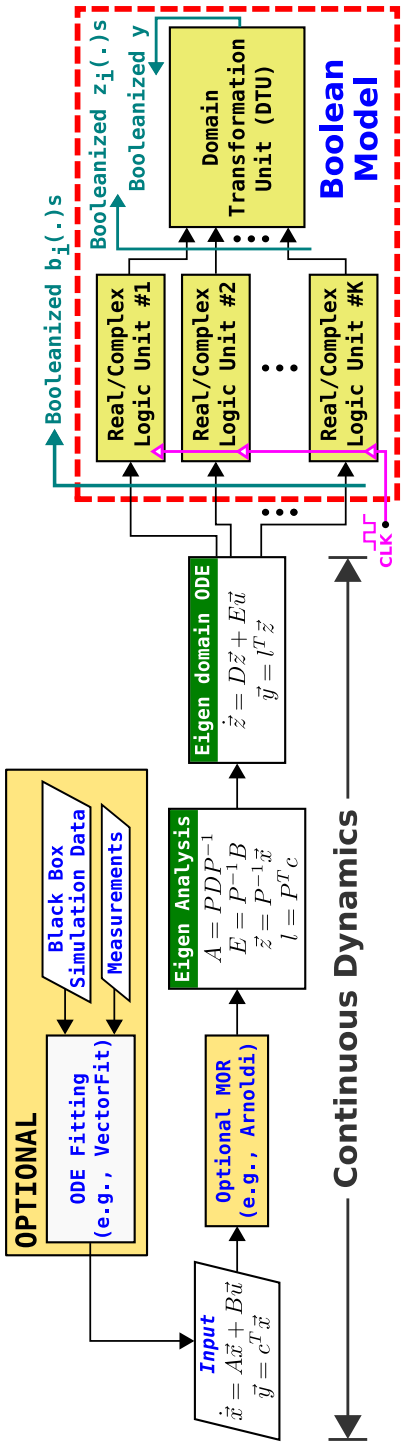


Figure 5.1: The ABCD-L flow for producing a Boolean approximation that captures the analog dynamics of an LTI system. As shown in the figure, ABCD-L can also accept black-box simulation data as input, and it also integrates well with LTI MOR techniques.

Clearly, we can write a matrix-vector product equation like the above for every eigenvector/eigenvalue pair $(\vec{p}, \lambda)$ of A. Writing all these matrix-vector product equations, and consolidating them into a matrix-matrix product equation, we obtain the following:

$$AP = PD, \tag{5.5}$$

where P is a matrix whose columns are the eigenvectors of A, and D is a diagonal matrix containing the eigenvalues in the same order [56]. Assuming that P is invertible (this is almost always true, both in a mathematical (Lebesgue measure) sense and in a practical (LTI systems that typically occur in the real-world) sense), and post-multiplying both sides of the equation above by P^{-1} (the matrix inverse of P), we obtain the following:

$$A = PDP^{-1}. \tag{5.6}$$

Substituting the above into Eq. (5.3), we get the following:

$$\begin{aligned} \frac{d}{dt}\vec{x} &= PDP^{-1}\vec{x} + B\vec{u}, \text{ and} \\ \vec{y} &= c^T\vec{x}. \end{aligned} \tag{5.7}$$

Pre-multiplying both sides of the first equation above by P^{-1} gives us the following:

$$\begin{aligned} P^{-1}\frac{d}{dt}\vec{x} &= DP^{-1}\vec{x} + P^{-1}B\vec{u}, \text{ and} \\ \vec{y} &= c^T\vec{x}. \end{aligned} \tag{5.8}$$

Since P^{-1} is constant with respect to time t, the operators P^{-1} and d/dt commute; so we can rewrite the equation above as follows:

$$\begin{aligned} \frac{d}{dt}P^{-1}\vec{x} &= DP^{-1}\vec{x} + P^{-1}B\vec{u}, \text{ and} \\ \vec{y} &= c^T\vec{x}. \end{aligned} \tag{5.9}$$

Writing $c^T\vec{x}$ as $c^TPP^{-1}\vec{x}$ in the second equation above, we get the following:

$$\begin{aligned} \frac{d}{dt}P^{-1}\vec{x} &= DP^{-1}\vec{x} + P^{-1}B\vec{u}, \text{ and} \\ \vec{y} &= c^TPP^{-1}\vec{x}. \end{aligned} \tag{5.10}$$

Denoting $P^{-1}\vec{x}$, $P^{-1}B$, and P^Tc by $\vec{z}$, E, and l respectively, we obtain the following new "eigen-ODE" system:

$$\frac{d}{dt}\vec{z} = D\vec{z} + E\vec{u}, \text{ and}$$
$$\vec{y} = l^T\vec{z}. \tag{5.11}$$

The equations relating the matrices ($A,\ B,\ c$) and ($D,\ E,\ l$) (through the eigenvector matrix P), collected in one place, can be found in Fig. 5.1).

Clearly, the i^{th} equation of the new ODE (Eq. (5.11)) is a "de-coupled" scalar linear differential equation that takes the following form:

$$\frac{d}{dt}z_i = \lambda_i z_i + b_i(t), \tag{5.12}$$

where λ_i is the i^{th} diagonal entry $D_{i,i}$ of the matrix D and $b_i(.)$ is the i^{th} entry of the vector $E\vec{u}(.)$. Given the initial condition $z_i(t_0)$, the solution to the above equation is known analytically, and is given by the following:

$$z_i(t) = z_i(t_0)e^{\lambda_i(t-t_0)} + \int_{t_0}^{t} b_i(\tau)e^{\lambda_i(t-\tau)}d\tau. \tag{5.13}$$

At this point, we would like to make some observations:

- On some (extremely rare, Lebesgue measure zero) occasions, the given LTI system may not be diagonalizable, *i.e.*, the matrix D above, instead of being diagonal, may take a Jordan form. It is possible (though tedious) to develop a general theory for Booleanizing such systems; however, because this almost never happens in practice, we do not consider it in this chapter.

- In some cases, the size of the given system makes eigen-analysis impractical. In such situations, we first use an LTI MOR technique (*e.g.*, Arnoldi iteration [40,57]), to reduce the system size, and then subject the reduced system to eigen-analysis. The theory of LTI MOR is well-developed, and many large LTI systems encountered in practice can successfully be reduced using the MOR techniques available today. Also, as Fig. 5.1 shows, it is relatively straightforward to integrate virtually any MOR technique into the ABCD-L flow; we demonstrate this for a 25849-node benchmark power grid network made available by IBM, in Section 5.6.

- Many real-world LTI systems are stiff, *i.e.*, their underlying signals evolve at widely different timescales because the systems' eigenvalues span several orders of magnitude. ABCD-L can efficiently handle such systems by using different timescales to Booleanize the dynamics corresponding to different eigenvalues. However, for the sake of notational and expositional simplicity, we do not elaborate on this here.

Resuming the ABCD-L flow, the key idea behind ABCD-L is to *transform* the analytical solution of Eq. (5.13) into *Boolean operations* that can be expressed using digital logic constructs (registers, counters, *etc.*).

This transformation is achieved by a set of purely Boolean *Logic Units (LUs)*. As Fig. 5.1 shows, the Boolean model produced by ABCD-L contains one such logic unit (LU) for each entry in $\vec{z}$. Thus, the returned Boolean model contains as many LUs as the size of the LTI system being Booleanized.

Each of the above LUs is either a *real LU (RLU)* or a *complex LU (CLU)*, depending on whether the corresponding eigenvalue (λ_i using the notation in Eq. (5.12)) is real or complex respectively. (For more details on how these LUs are constructed, please see the discussion below.).

Furthermore, as Fig. 5.1 shows, the output sequence produced by the i^{th} such LU is a (multi-bit) Boolean approximation to the i^{th} entry of $\vec{z}(t)$ (z_i, using the notation in Eq. (5.12)). That is, if one collects the (discrete) symbol sequence produced by the i^{th} LU and maps this symbol sequence back into the continuous domain (please see Section 3.4 for a discussion on how symbol sequences can be interpolated back into continuous waveforms), one would obtain a waveform that approximates the solution $z_i(t)$ predicted by Eq. (5.13).

We now return back to how the Boolean model produced by ABCD-L is organized (Fig. 5.1). In addition to the LUs, as Fig. 5.1 shows, the Boolean model also contains a so-called *Domain Transformation Unit*, or *DTU*. This DTU is a purely combinational logic unit: its function is to *combine* the outputs of the LUs into a multi-bit Boolean approximation of the circuit's outputs $\vec{y}(t)$. Recall that the LUs produce approximations to the z_is at their outputs, and that the LTI system's overall output $\vec{y}$ is related to the z_is by the equation $\vec{y} = l^T\vec{z}$. Therefore, in essence, the DTU just implements a Booleanized approximation of the equation $\vec{y} = l^T\vec{z}$. Since this equation is purely algenraic (*i.e.*, it is not a differential equation), the DTU can be purely combinational (*i.e.*, it does not have to be sequential).

Finally, the outputs $\vec{y}(t)$ produced by the DTU above can again be mapped back into the continuous domain, and compared against the responses produced by the original LTI system for similar input waveforms (as described in Section 3.4). Indeed, the examples in the next few sections all carry out this post-processing step to demonstrate the accuracy of the Boolean models generated by ABCD-L.

Thus far, we have focused on the *high-level organization* of the Boolean model produced by ABCD-L (Fig. 5.1). We will now delve into lower level details, such as the techniques used by ABCD-L to structure and construct the building blocks of the Boolean model in Fig. 5.1 – namely, the DTU and the individual LUs.

We begin with the DTU because, being purely combinational, it is simpler to understand how it should be constructed. Indeed, since the DTU simply needs to implement a Booleanized version of the equation $\vec{y} = l^T\vec{z}$, where the inputs to the DTU are discretized versions of $\vec{z}$, the DTU's output is simply a Booleanized linear combination of its inputs. This is a purely combinational function whose Boolean specification/synthesis has been well-studied (*e.g.*, see [58]). One way to realize this combinational function is to use adders and multipliers to essentially mimic the matrix pre-multiplication by l^T within the DTU.

However, using multipliers is *not* recommended because verification problems involving multipliers are most often computationally intractable, and using them may severely limit the scalability of the resulting Boolean model for AMS verification. Therefore, ABCD-L adopts what we call a 2-*step discretization process* for implementing the DTU (described below).

Let us say we wish to compute the i^{th} entry of the output $\vec{y}$, namely, y_i. From the equation above, this is given by:

$$y_i = l_{1,i}\ z_1 + l_{2,i}\ z_2 + \ldots + l_{n,i}\ z_n,$$

where n denotes the size of the given LTI system. The key idea is that we do not calculate the individual terms in the sum above by implementing a multiplier in hardware. Instead, we use a *finer discretization* to compute the individual terms above. That is, if the output y_i is to be discretized using 4 bits, we may discretize each of the terms above using 8 bits or even higher. In this way, instead of attempting to do a floating point like multiplication to calculate the terms above, we just maintain a small truth table (or BDD, or AIG) per term. For example, to calculate the j^{th} term in the sum above, the j^{th} truth table (or BDD, or AIG) would take as input a discretized version of z_j (which is usually about 4 to 8 bits wide). Given this input, the j^{th} truth table (or BDD, or AIG) would produce as output a finely discretized approximation to $l_{j,i}z_j$, which is typically about 8 to 16 bits wide. These truth tables (or BDDs, or AIGs) are typically small enough that they are not expensive to maintain, and they usually produce acceptably accurate Boolean models without needing to carry out expensive and hard to analyze/verify floating point multiplications. The outputs of these truth tables (or BDDs, or AIGs), can then be combined to produce a discrete version of y_i using some more combinational logic.

This completes the discussion of the DTU. We now describe the structuring of the real and the complex LUs.

The LUs, as opposed to the DTU, are sequential systems, and their construction is more challenging. Each LU implements, using purely Boolean logic, a scalar linear differential equation that takes the following form:

$$\frac{d}{dt}z_i(t) = \lambda_i z_i(t) + b_i(t). \tag{5.14}$$

The signals $z_i(t)$ and $b_i(t)$ are encoded as bit-vectors of length m (where m, as described before, is a parameter passed to ABCD-L by the user, called the *signal resolution*). The LU is designed so that its bit-vector approximation to $z_i(t)$, when mapped back into the analog domain, closely matches the actual system response $z_i(t)$ for all input sequences $b_i(t)$. This is achieved by the logic structures depicted in Figs. 5.2 and 5.3, for real and complex eigenvalues respectively.

Let us first consider the real eigenvalue scenario, *i.e.*, the problem of Booleanizing the differential equation above, where all quantities are real. Fig. 5.2 shows a Boolean schematic for this, which includes (a) a *Signal Register* `SR` that maintains an m-bit representation of z_i, (b) a 1-bit *Direction Register* `DR`, that represents whether z_i is increasing or decreas-

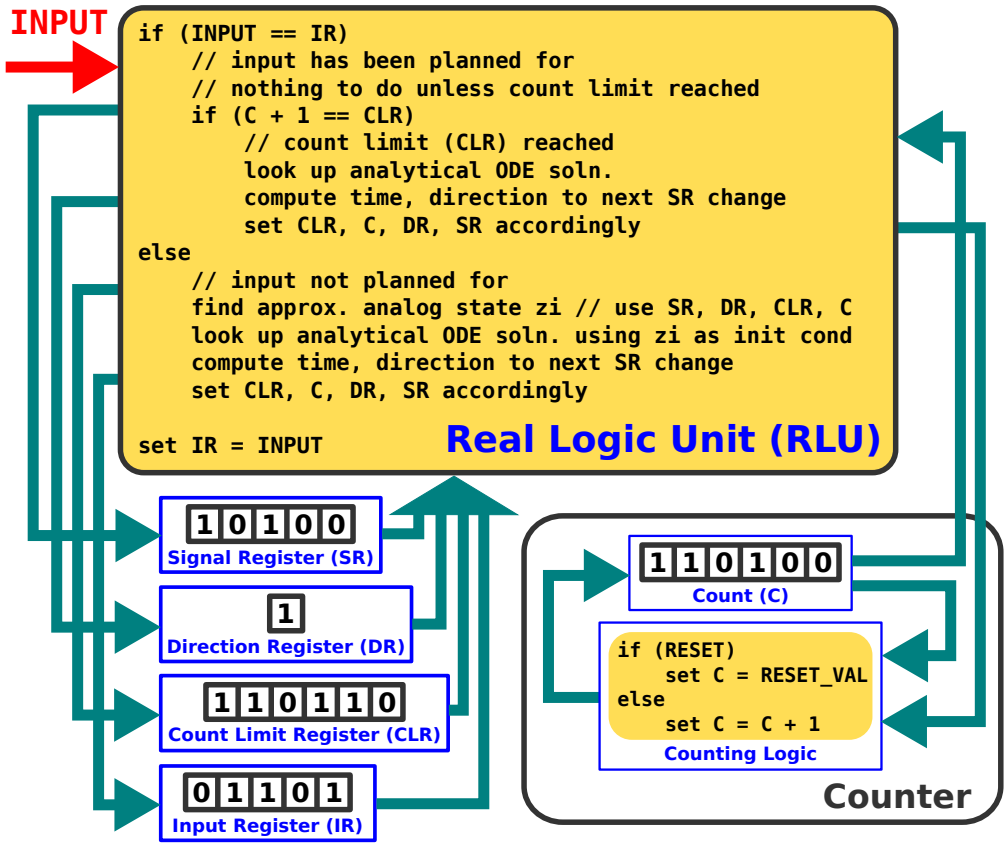


Figure 5.2: Sequential logic schematic for discretizing a real scalar linear differential equation $\frac{d}{dt}z_i = \lambda_i z_i + b_i(t)$ in the eigen-domain.

ing, (c) a *Count Limit Register* `CLR` that indicates the time at which `SR` must be incremented/decremented, (d) a set/reset counter with a count `C`, which measures time by counting up to the limit `CLR`, and (e) an m-bit *Input Register* `IR` that stores the input b_i from the previous clock cycle. As described before, the whole unit is clocked at a pre-determined, fixed time-step Δ (in practice, it is usually straightforward to choose Δ to ensure that it is small enough to capture the dynamics of the scalar system above). For stiff systems, one can boost computational efficiency by using different Δs for the different LUs.

The above unit works as follows: as long as the input $b_i(t)$ remains constant, it is, in some sense, already "planned for". That is, the above structure stores enough information to know when, and in which direction, the register `SR` must be incremented/decremented. In this case, the count `C` keeps ticking up until it eventually becomes equal to the count limit `CLR`, at which time the register `SR` (and all the other registers as well) are updated accordingly. Analytical expressions, based on Eq. (5.13), are known for the "time to next change in `SR` mod Δ" operation, which can be either computed on the fly, or stored in a truth-table, *etc.*.

For non-constant $b_i(t)$, as Fig. 5.2 indicates, the logic unit responds by considering the latest sampled input to be a new DC input, and updates all the registers using the analytical "time to next change mod Δ" function, making an intelligent estimate about the current value of z_i using the contents of registers `SR`, `C`, `CLR`, and `DR` (note that the count can be reset to any value by passing the `RESET` signal to the counter).

For concreteness, let us consider an example. Assume that our LU Booleanizes the equation $dz_i/dt = -10^9 z_i + 10^9 b_i(t)$. Let us assume that z_i and b_i are represented using 4 bits each in our Boolean model, *i.e.*, `SR` and `IR` each has a width of 4, with 0000 (1111) representing 0V (1V). Since the time-constant of this system is clearly 1ns, let us conservatively clock the LU at 1/50ns. Suppose `SR` currently reads 0000 (*i.e.*, $z_i \approx$ 0V), and the input b_i suddenly swings from 0V to 1V. From Eq. (5.13), we know that $z_i(t)$ will reach 1/15V (*i.e.*, `SR` will become 0001) in about 0.069ns, or about 3 clock cycles. So, on the positive clock edge immediately after the input switch, the count `C` would be reset to 0000 and `DR` and `CLR` would assume the values 1 (to denote that z_i is increasing) and 0011 (to represent a delay of 3 clock cycles) respectively. If b_i continues to remain at 1.0V, then, in future clock cycles, the count `C` will be continually incremented until it reaches the `CLR` value 0011, at which time it will be reset back to 0000, `SR` will be incremented to 0001, and `CLR` will be updated to 0100 (to indicate a further delay of 4 clock cycles before z_i reaches the next threshold of 2/15V).

This is an implementation of the "pattern" of introducing additional FSM states (that are traversed successively by means of a counter) to model analog delays; we will also use this pattern extensively in developing ABCD-NL (Chapter 6). This pattern helps ensure that the resulting Boolean model does not contain undesirable artifacts such as "fake fixed points" (described in detail in Section 6.3.2); these are stable states/cycles in the Boolean model, introduced as undesirable artifacts of the process of Booleanization, that do not exist in the original ODE/DAE. Boolean models produced by ABCD-L, unlike those produced by more naïve methods, are guaranteed not to have such artifacts.

For the complex eigenvalue case, a CLU (Fig. 5.3) essentially consists of two copies of each RLU register, storing the real and imaginary parts of each underlying signal. Whenever the registers need to be updated (for example, if the input has changed or if the limit `CLR` is reached by one of the counters), both the real and the imaginary sets of registers are updated simultaneously, based on the analytical solution given by Eq. (5.13).

From the above discussion, ABCD-L's accuracy clearly increases with the signal resolution m (which is a designer-specified parameter that determines how finely the underlying signals are discretized). In principle, this allows ABCD-L to abstract the given LTI system to any desired level of accuracy. In practice, for a designer, it is usually straightforward to determine an appropriate value for m through trial and error.

We also note that time-domain simulations involving ABCD-L's Boolean models can be very efficient, because they can be carried out entirely in the logical/Boolean domain (if necessary, using specialized logic simulation tools), without requiring any differential equation solving. To speed up simulation even further, we can devise an algorithm that jumps directly to the time instant specified by `CLR`, instead of incrementing the count `C` at every

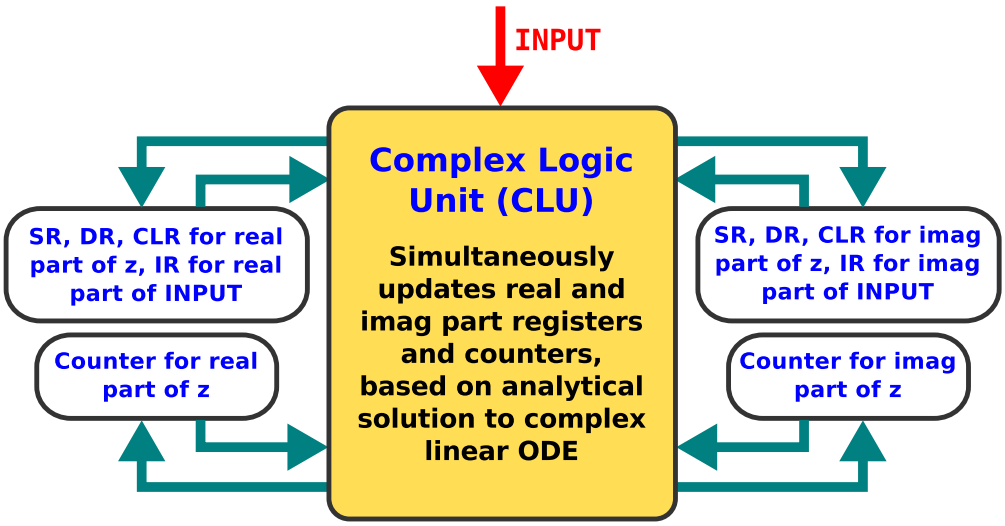


Figure 5.3: Sequential logic implementation schematic for discretizing a complex scalar linear differential equation $\dot{z}_i = \lambda_i z_i + b_i(t)$ in the eigen-domain.

time-step. Indeed, we have implemented this algorithm, and in Section 5.7, we demonstrate that it can be significantly faster than conventional circuit simulation methods such as linear multi-step integration (even after accounting for the time taken up by eigen-analysis, model generation, *etc.*).

Having described the core techniques behind ABCD-L, we now apply them towards Booleanizing LTI systems that are of interest to AMS designers, including, (1) I/O links modelled using RC and RLGC chains, (2) an "LTI channel followed by a differential equalizer" circuit, linearized using small-signal analysis, and (3) A 25849-node power-grid circuit, Booleanized by applying ABCD-L in conjunction with LTI Model Order Reduction (MOR) techniques. In each case, we show that the Boolean models produced by ABCD-L are able to accurately capture the continuous-time dynamics of such systems, including important analog effects such as inter-symbol interference (ISI), overshoot/undershoot, ringing, *etc.*. Also, we show that ABCD-L is able to faithfully reproduce higher-level AMS-design metrics for such systems, including the *entire shape of the eye diagram opening* at key circuit nodes. Furthermore, from a computational efficiency perspective, we show that ABCD-L can offer considerable simulation speed-ups over traditional LTI ODE/DAE simulation methods like linear multi-step integration (especially for larger LTI systems).

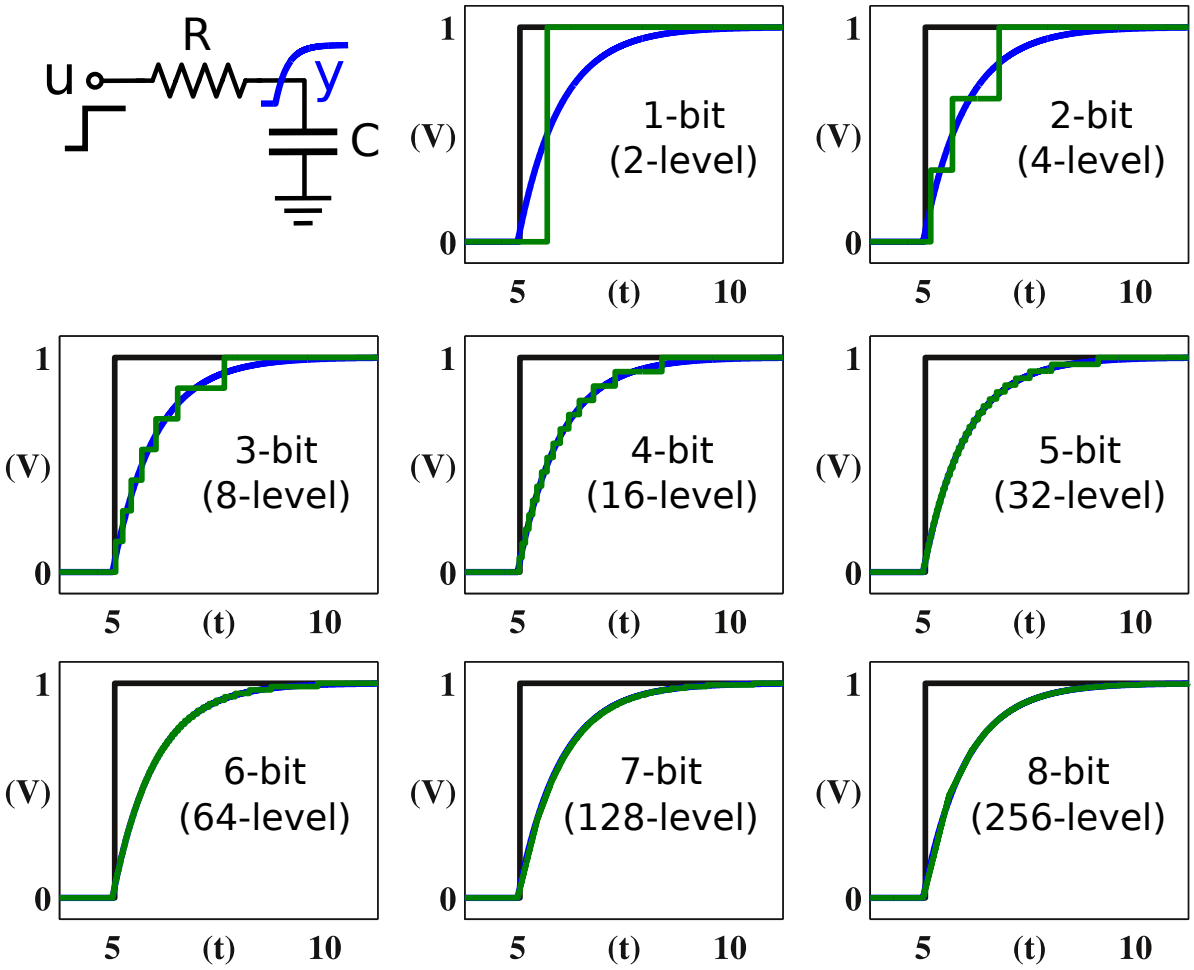


Figure 5.4: ABCD-L applied to an RC filter, to produce Boolean models of arbitrarily high accuracy. The system's response to a step input $u(t)$ (in black) is computed both analytically (graphed in blue), and by simulating ABCD-L-generated Boolean models (graphed in green). In all the plots above, the X-axis denotes time in RC units, and the Y-axis denotes voltages in Volts.

5.3 Example: RC/RLGC filters and arbitrarily high Booleanization accuracy

Before presenting the examples described above, we would first like to highlight an important feature offered by ABCD-L: in spite of using purely Boolean models, ABCD-L can reproduce the continuous-time dynamics of LTI systems with *arbitrarily high accuracy*, simply by increasing the number of bits used to represent the underlying circuit signals. We now illustrate this using two simple examples: an RC filter (Fig. 5.4), and an RLGC filter (Fig. 5.5).

Fig. 5.4 illustrates the application of ABCD-L to an RC filter (shown at the top left of the figure). This is a linear system of size 1, whose only eigenvalue is real. The system is described by the following differential equation:

$$C\frac{d}{dt}y + \frac{y-u}{R} = 0, \tag{5.15}$$

where $u(t)$ is the input to the filter and $y(t)$ is the filter's output.

As described in Section 5.2, ABCD-L represents the input $u(t)$ and the output $y(t)$, as bit vectors of length m, where higher values of m correspond to finer quantization of the underlying analog signals. In Fig. 5.4 above, we increase m (in steps of 1) from 1 to 8, and as we do so, the responses predicted by the ABCD-L-generated Boolean models (the green waveforms) resemble the analytically calculated responses (the blue waveforms) ever more closely. This indicates that the accuracy of ABCD-L rapidly increases with the signal resolution m. Furthermore, as mentioned above, as $m \to \infty$, this accuracy approaches exactness, making arbitrarily high accuracy possible.

Fig. 5.5 depicts the application of ABCD-L to an RLGC filter (shown at the top left of the figure), which again illustrates the point that Boolean models with arbitrarily high accuracy may be derived using ABCD-L for LTI systems. The RLGC filter of Fig. 5.5 is a linear system of size 2, whose eigenvalues are both complex. The DAE system representing this circuit takes the following form:

$$\begin{pmatrix} C & 0 \\ 0 & L/R \end{pmatrix} \frac{d}{dt} \begin{pmatrix} y \\ i_L \end{pmatrix} + \begin{pmatrix} G & -1 \\ 1/R & 1 \end{pmatrix} \begin{pmatrix} y \\ i_L \end{pmatrix} + \begin{pmatrix} 0 \\ -1/R \end{pmatrix} u(t) = \vec{0}, \tag{5.16}$$

where $u(t)$ is the input to the filter, $y(t)$ is the filter's output, and $i_L(t)$ is the time-varying current flowing through the inductor from left to right in the circuit shown in the top left portion of Fig. 5.5.

As described in Section 5.2, ABCD-L discretizes the circuit's input $u(t)$ (in this example, a unit step function), its internal voltages/currents, and its output $y(t)$, into bit vectors of length m, where higher values of m correspond to finer quantization of the underlying analog signals (hence greater accuracy). The figure shows that, as we increase m from 1 to 8, the response predicted by ABCD-L's Boolean model (the green waveform) matches the actual system's response (the blue waveform) more and more accurately, duplicating important

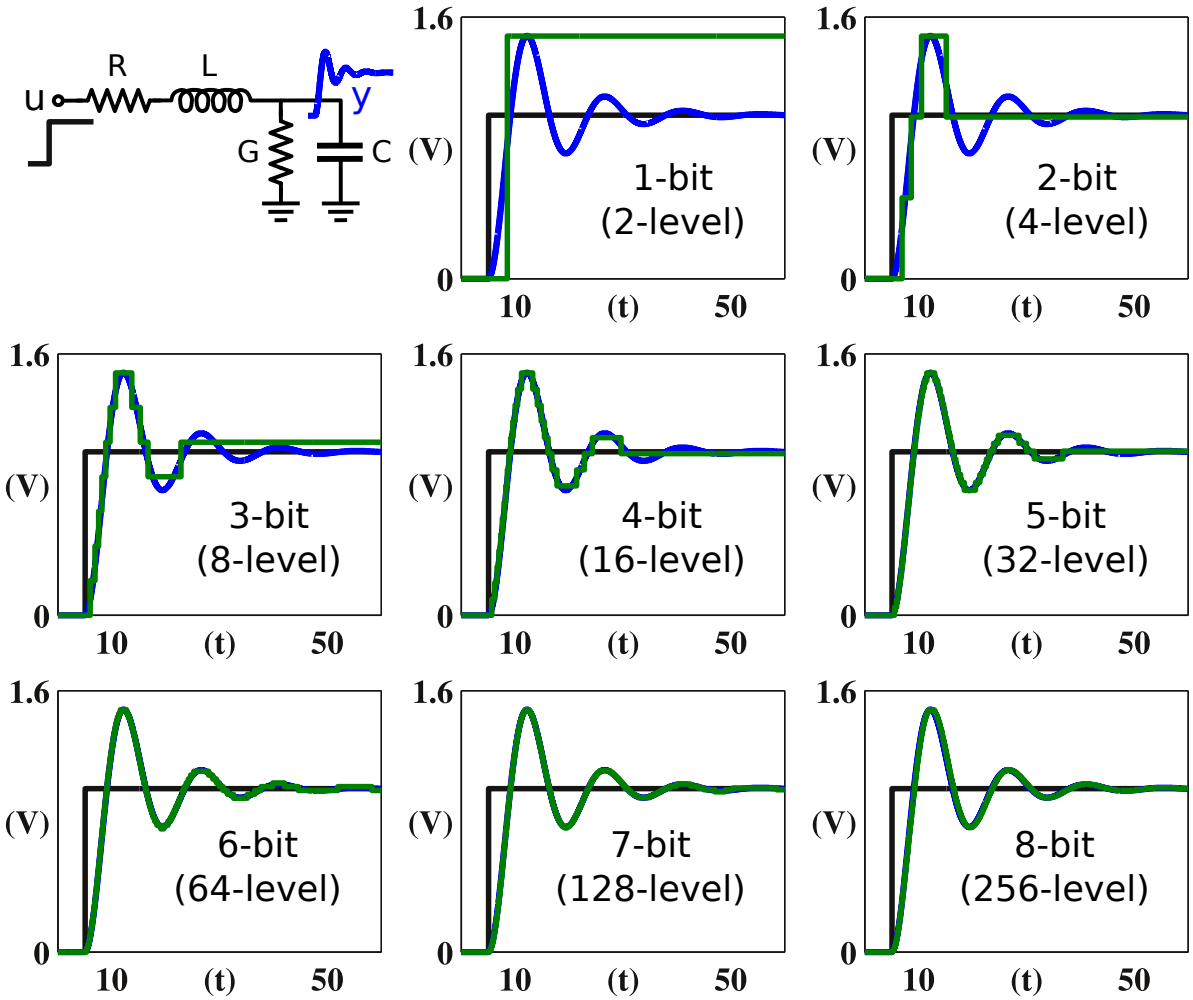


Figure 5.5: ABCD-L applied to an RLGC filter, to produce Boolean models of arbitrarily high accuracy. The continuous system's response to a step input $u(t)$ (in black) is computed both analytically (blue), and by simulating ABCD-L-generated Boolean models (green). In the plots above, the X-axis denotes time in RC units, and the Y-axis denotes voltages in Volts.

features such as overshoot and ringing. This again serves to illustrate a point that is true for LTI systems in general: ABCD-L's Boolean approximations can be made as close to the original system as desired, simply by increasing the number of bits used to discretize the underlying signals, as well as the clock frequency of the underlying Boolean model.

5.4 Example: Booleanizing RC and RLGC chains

We now apply ABCD-L to chains of RC and RLGC units (Fig. 5.6); these are often used to model high-speed I/O links, interconnect networks, on-chip communication channels, *etc.* [59, 60].

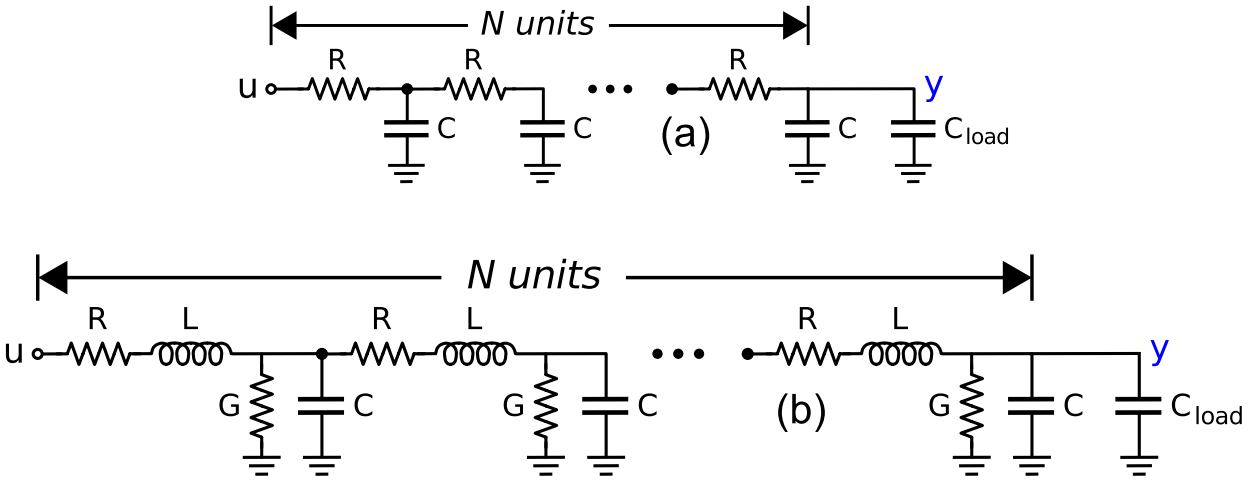


Figure 5.6: RC and RLGC chains of length N, each driving a load capacitance C_{load}

Our experiments on RC/RLGC chains run as follows:

- We build an RC/RLGC chain, and apply several long, randomly generated bit patterns $u(t)$ at its input. To model both small and large inter-symbol interference (ISI), we vary the unit-interval T (the time that elapses between successive bits at the input), which is the inverse of the bitrate. Note that ISI decreases with increasing T, and vice-versa.
- We use ABCD-L to predict the system's responses $y(t)$ to the above inputs. Between experiments, we vary ABCD-L's signal resolution parameter m (the number of bits used by ABCD-L to quantize the underlying circuit's eigen-domain signals).
- We compare ABCD-L's time-domain predictions against those of an ODE/DAE solver, plotting them on top of one another.

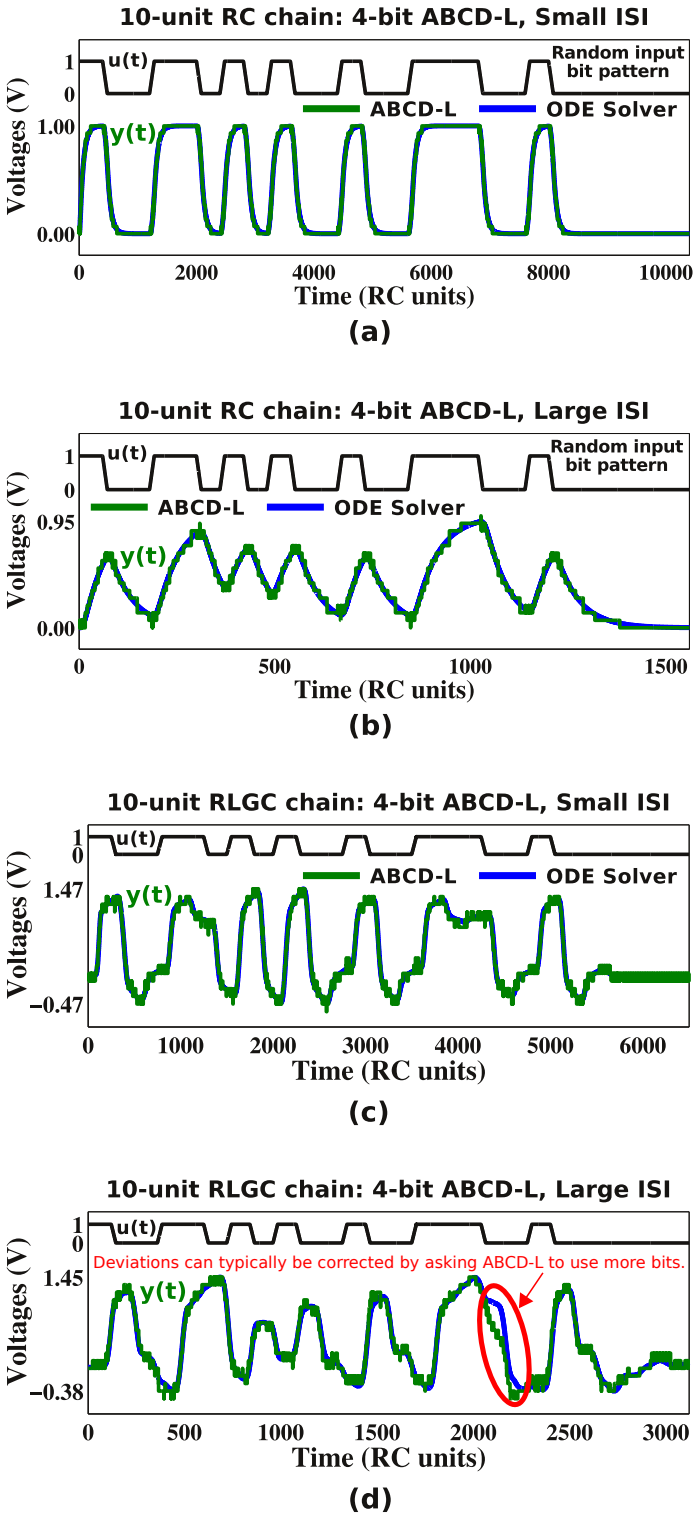


Figure 5.7: Applying 4-bit and 8-bit ABCD-L to 10-unit RC and RLGC chains. The resulting Boolean models are simulated under conditions of both small ISI (parts (a), (c)) and large ISI (parts (b), (d)), and the Boolean models' predictions (the green waveforms) are compared against actual system responses (the blue waveforms), for a randomly generated input pattern (the black waveforms labelled $u(t)$).

- Finally, since I/O link/interconnect engineering often makes extensive use of eye diagrams [61], we also convert ABCD-L's predictions into eye diagram form, and compare against eye diagrams generated by an ODE/DAE solver.

Fig. 5.7 depicts the results obtained by applying 4-bit ABCD-L to a 10-unit RC chain, and to a 10-unit RLGC chain, under conditions of both small ISI and large ISI. We note that, because all signals are quantized using 4 bits, the output of the system, as predicted by ABCD-L, consists of at most $2^4 = 16$ different levels. Moreover, as seen from the figure, in all cases, the 16-level predictions made by ABCD-L (drawn in green) closely match the system's actual continuous-time responses (drawn in blue). Therefore, the Booleanized models produced by ABCD-L appear to be good approximations of the underlying continuous LTI systems.

In parts (a) and (c) of Fig. 5.7, the bitrate of the applied input (the black waveform labelled $u(t)$) is low enough that the resulting ISI is small. This is readily seen from the system's responses: whenever the input bit is high (low), the output response has enough time to rise (fall) to a reasonably high (low) level before the next bit arrives. On the other hand, in parts (b) and (d) of the figure, we increased the input bitrates to a point where the induced ISI became significant. This can also be seen visually from the figure – even if an input bit is high (low), the output responses often do not have enough time to rise (fall) before the next bit arrives. As the figure shows, all these effects are captured quite accurately by ABCD-L.

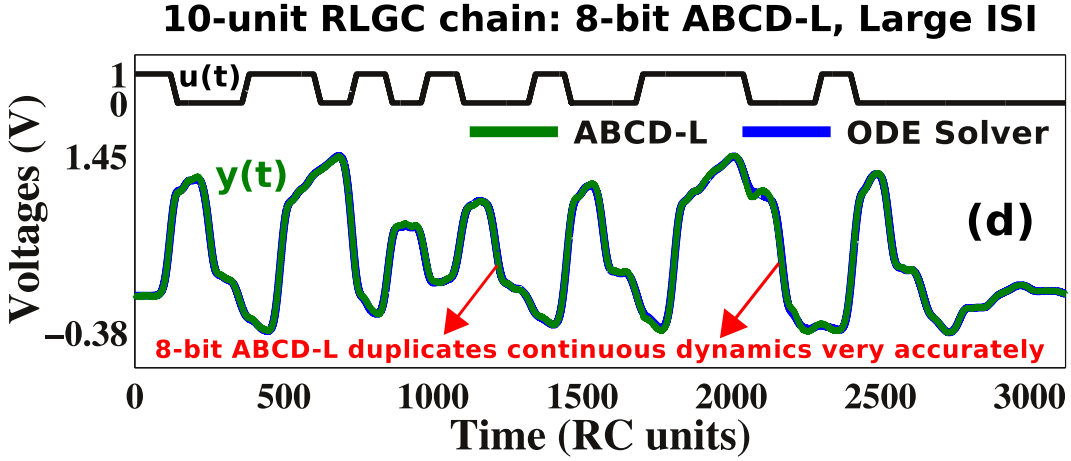


Figure 5.8: Applying 8-bit ABCD-L to a 10-unit RLGC chain, for the same inputs as in Fig. 5.7(d). With increased signal resolution, deviations between ABCD-L's prediction and the system's actual response are significantly reduced.

In Fig. 5.7(d), we have drawn attention (with a red circle) to a small time-interval where there is some deviation between ABCD-L's prediction and the system's response. Although such deviations tend to be "self-correcting" (as seen from the figure), it is desirable to

reduce the magnitude of these deviations. As described in Section 5.2, this can typically be achieved simply by increasing the number of bits used by ABCD-L to discretize the underlying waveforms (this corresponds to changing a single parameter that is passed as an argument to ABCD-L). This is illustrated in Fig. 5.8, where we have applied 8-bit ABCD-L (instead of 4-bit ABCD-L) to the same RLGC chain, for the same inputs as in Fig. 5.7 (d). From the figure, it is readily seen that the deviations between ABCD-L and the original system have been significantly reduced. This supports our earlier assertion that ABCD-L can model any LTI system with arbitrarily high accuracy.

Having illustrated ABCD-L's ability to closely match the underlying system's responses for short input patterns, we now simulate the ABCD-L-generated Boolean models on much longer input patterns (thousands of bits), and convert the resulting predictions into *eye diagram form*; this representation is extensively used in I/O link/interconnect design, modelling, analysis, and simulation, because it provides the design architect with a quick snapshot of key system properties [61].

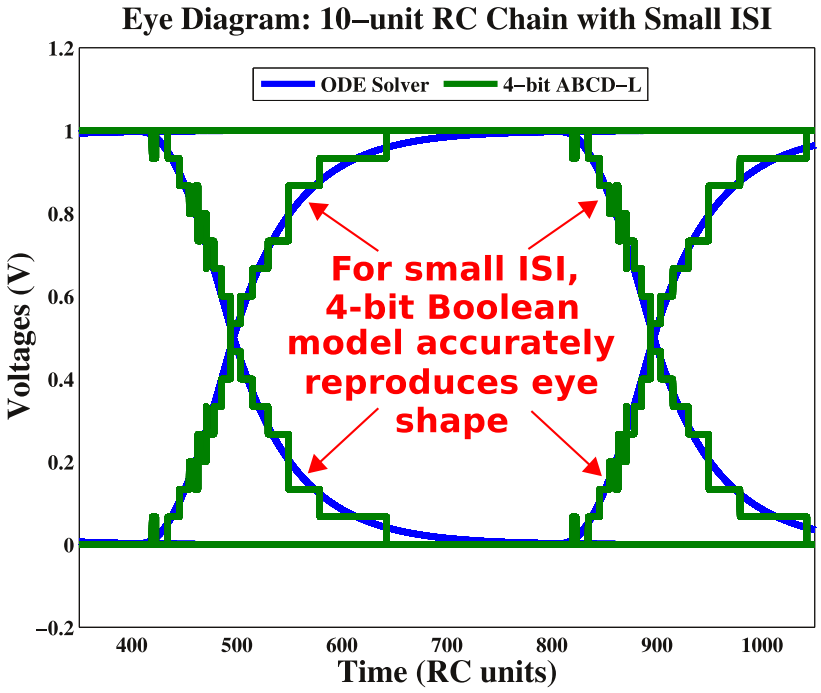


Figure 5.9: Eye diagrams produced by 4-bit ABCD-L (green), and by an ODE solver (blue), for a 10-unit RC chain under conditions of small ISI.

Fig. 5.9 depicts the eye diagrams produced by both 4-bit ABCD-L (in green), and by an ODE solver (in blue), for the 10-unit RC chain of Fig. 5.6, under conditions of small ISI. As is to be expected for small ISI, this eye diagram has a wide opening, which at times extends all the way from 0V to 1V. Also, it is clear from the figure that the 4-bit Boolean model is able to reproduce the shape of the eye opening with good accuracy.

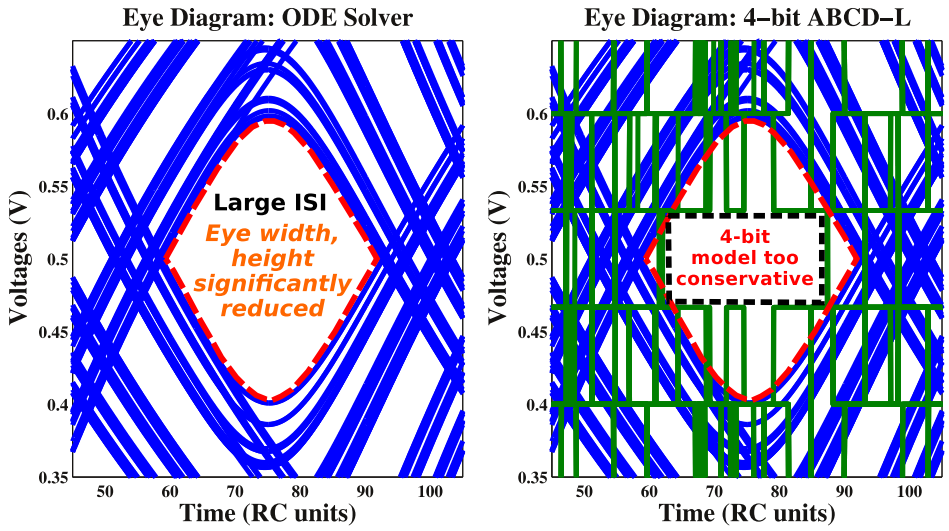


Figure 5.10: Eye diagrams produced by 4-bit ABCD-L (green, right), and by an ODE solver (blue, left and right), for a 10-unit RC chain under conditions of large ISI.

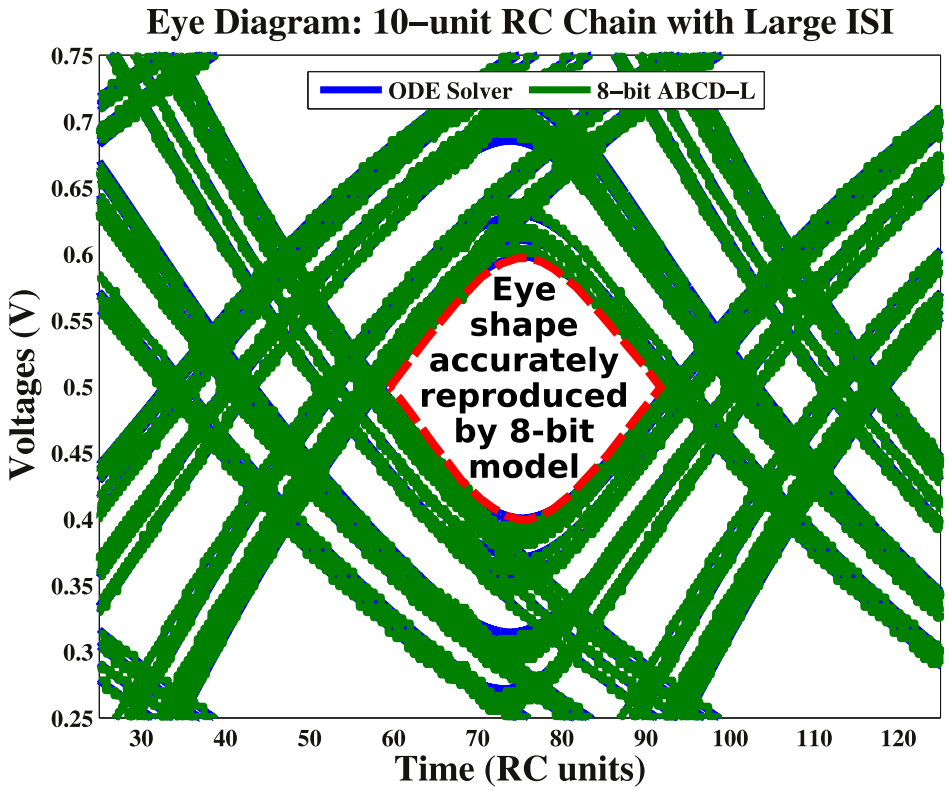


Figure 5.11: Eye diagrams produced by 8-bit ABCD-L (green), and by an ODE solver (blue), for a 10-unit RC chain under conditions of large ISI.

Fig. 5.10 (left) depicts the shape of the eye opening (the red dashed contour) for the RC chain of Fig. 5.6, under conditions of large ISI. As the figure shows, due to increased ISI, the eye opening now extends only between ~ 0.4V and ~ 0.57V, even though the RC chain's input switches between 0V and 1V. Also, this range is now too small to be resolved accurately by 4-bit ABCD-L; the error due to discretization is significant. Indeed, as the green signal transitions in the right half of the figure show, the distance between successive quantization levels of $y(t)$ is too large to resolve $y(t)$ accurately. As a result, the 4-bit ABCD-L model produces an overly conservative eye opening (the black dashed line). For increased accuracy, it is necessary to use ABCD-L with a higher signal resolution (*e.g.*, 8 bits). Indeed, as shown in Fig. 5.11, the 8-bit ABCD-L model almost perfectly reproduces the shape of the eye opening under large ISI, *i.e.*, the contour defined by the green waveforms closely matches the red dashed contour obtained by ODE simulation. This confirms our intuition that, in general, as ISI (which is a measure of the linear system's inherent *memory*) increases, the signal resolution of ABCD-L also has to be increased, so as to preserve the accuracy of the generated Boolean models.

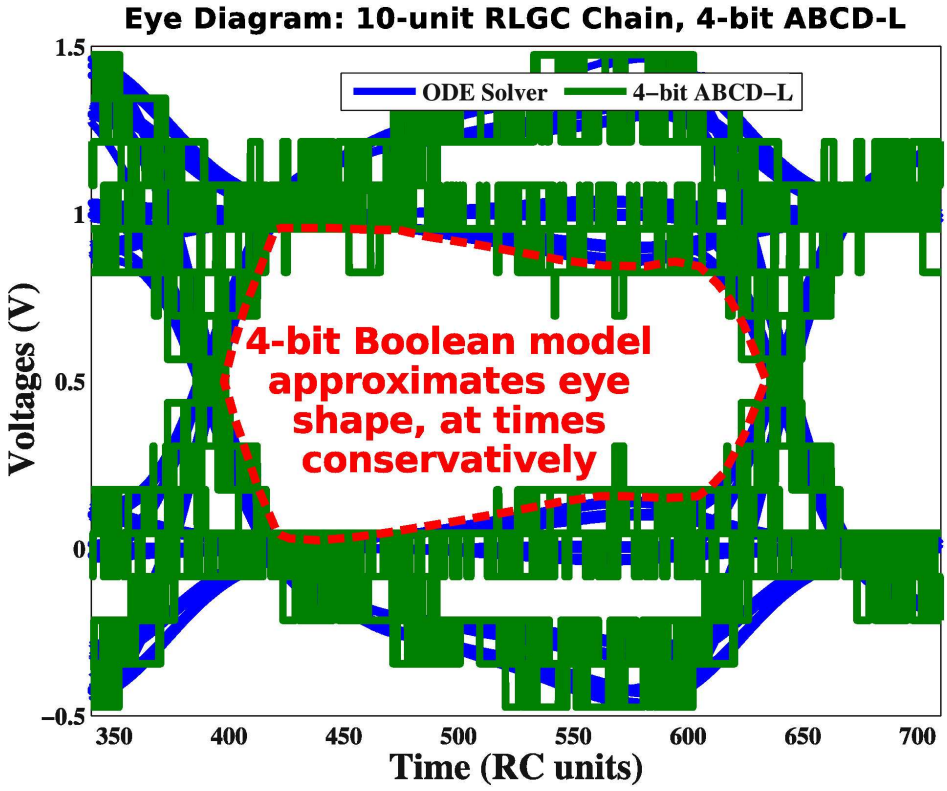


Figure 5.12: Eye diagrams produced by 4-bit ABCD-L (green), and by an ODE solver (blue), for the 10-unit RLGC chain.

Fig. 5.12 depicts the eye diagram produced by applying 4-bit ABCD-L (in green) to the 10-unit RLGC chain of Fig. 5.6. This eye diagram is overlaid on top of the eye diagram

produced by an ODE solver (which is in blue). The red dashed contour traces the shape of the eye opening, as predicted by the ODE solver. As seen from the figure, ABCD-L's 4-bit Boolean model is able to reproduce the entire shape of the eye opening with good accuracy.

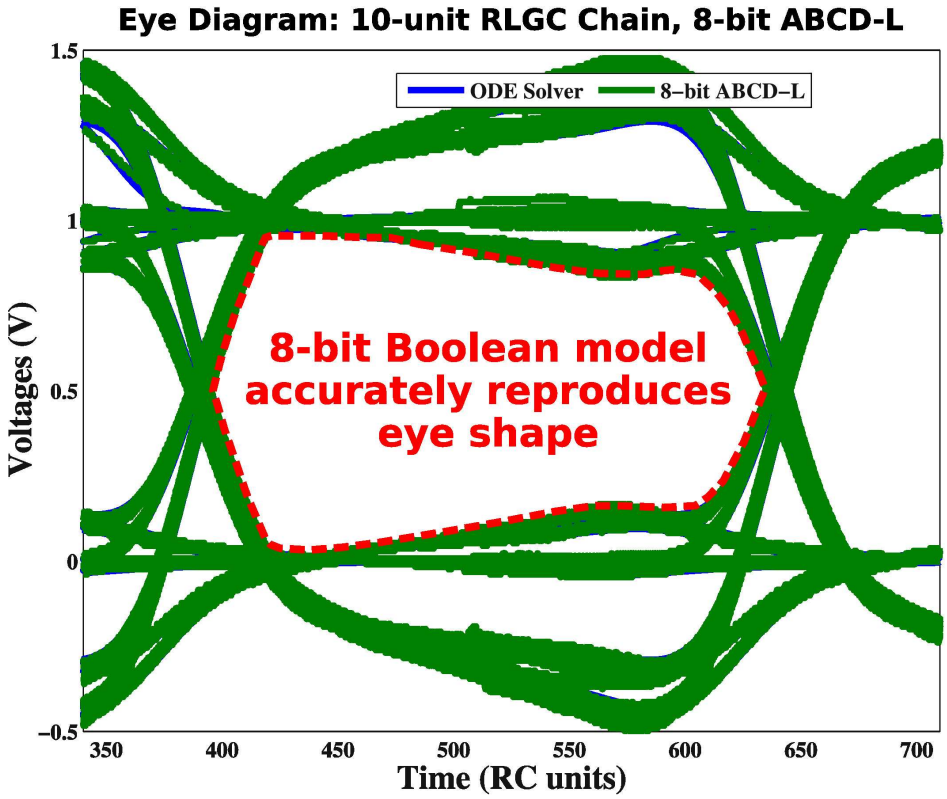


Figure 5.13: Eye diagrams produced by 8-bit ABCD-L (green), and by an ODE solver (blue), for the 10-unit RLGC chain. It is seen that the shape of the eye opening is reproduced with increased accuracy compared to Fig. 5.12.

However, the eye opening produced by 4-bit ABCD-L is at times a bit conservative. While this may not be a problem for many applications, it may sometimes be necessary to obtain a more accurate representation of the eye opening. As we have indicated before, such increased accuracy can be achieved simply by asking ABCD-L to use more bits to discretize the underlying circuit's signals. This is illustrated in Fig. 5.13, which depicts the eye diagram produced by 8-bit ABCD-L for the same RLGC chain. Indeed, as seen from the figure, the 8-bit ABCD-L model almost perfectly reproduces the shape of the entire eye opening (as compared to the red dashed contour obtained by ODE simulation). This also supports our assertion that ABCD-L can approximate LTI systems to any desired level of accuracy.

5.5 Example: Booleanizing a Channel + Equalizer circuit

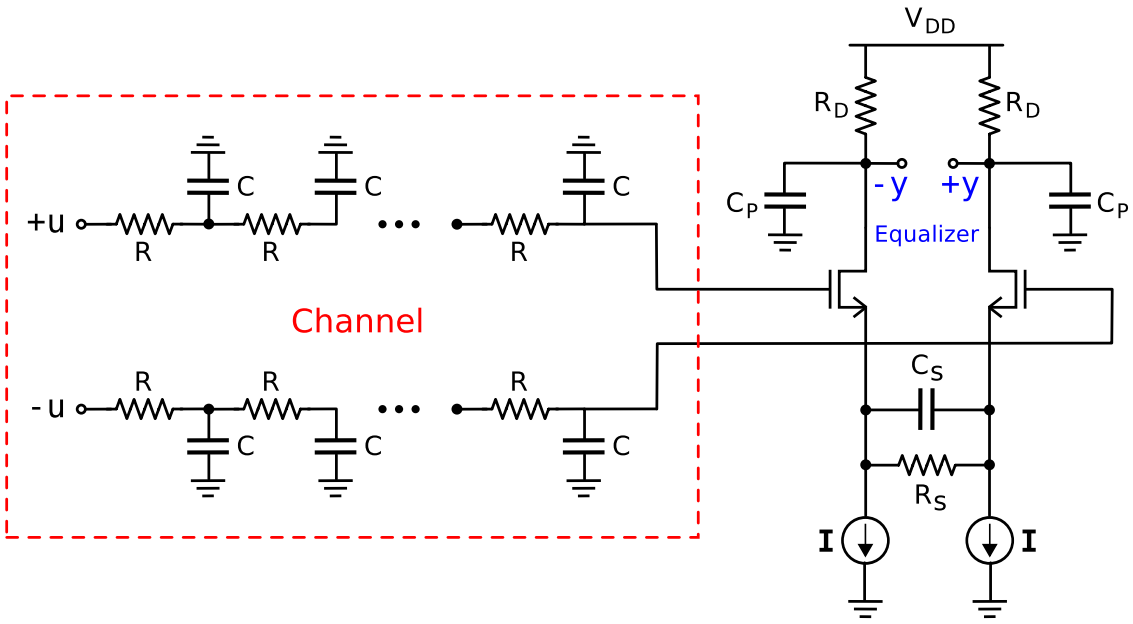


Figure 5.14: LTI channel followed by a differential equalizer.

Having applied ABCD-L to RC/RLGC chains, we now consider a more complex example relevant to AMS-design: an LTI channel followed by an equalization circuit, as illustrated in Fig. 5.14. We note that, in this circuit, both the channel (modelled as an RC chain) and the equalizer use *differential signalling*, *i.e.*, the circuit's inputs and outputs are represented by the *difference* between two voltages (instead of a single voltage).[3] The equalizer plays a critical rôle in this circuit: it *partially reverses* the distortion (ISI) produced by the channel, so that one can transmit bits across the channel at much higher speeds than would be possible otherwise. For example, if the channel's cut-off frequency is 1 GHz, then reliable transmission can happen only at bitrates at or below 1Gbps. However, if the combined "channel plus equalizer" system has an effective cut-off frequency at 3 GHz, then one can triple the throughput without suffering dispersion.[4]

We also note that the circuit in Fig. 5.14 is non-linear. Therefore, we apply ABCD-L not to the original circuit, but to a small-signal linearization of the original circuit around its quiescent operating point, using small-signal device models (obtained by applying the

[3]Differential signalling has important advantages over single-ended signalling, including better noise resilience, improved resistance to external interference, *etc.*.

[4]*Dispersion* is the term used to describe the degradation of the transmitted signal due to inter-symbol interference arising from *linear* circuit dynamics, while *distortion* is the term used to describe signal degradation due to *non-linear* circuit dynamics.

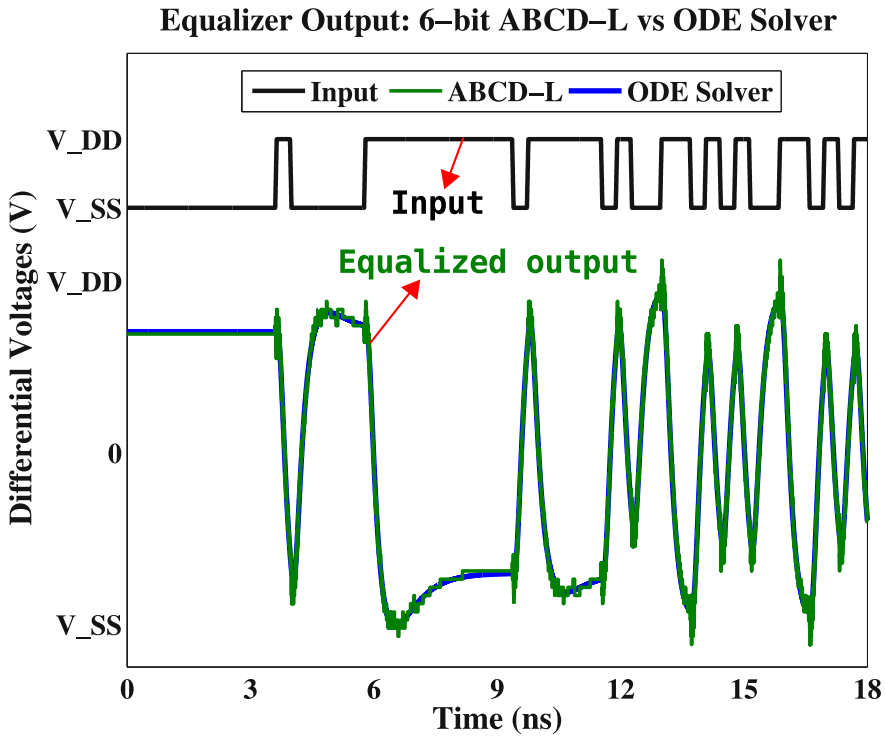


Figure 5.15: ABCD-L accurately reproduces the time-domain continuous dynamics of the equalizer.

techniques described in the book by Sedra and Smith [62]). In this case, linearization does not result in appreciable loss of accuracy because, the equalizer in Fig. 5.14, under the operating conditions we are interested in, was designed to behave approximately linearly.

Fig. 5.15 illustrates the application of 6-bit ABCD-L to the small-signal linearized "channel plus equalizer" circuit (henceforth simply referred to as "the system") above (with the channel being a 5-unit RC chain). The blue waveform of Fig. 5.15 was obtained by using an ODE solver to simulate the system, for a random bit pattern applied at the input (the black waveform). As before, we see from the figure that the Boolean model produced by ABCD-L is able to accurately duplicate the time-domain behaviour of the system.

For equalizers, an important AMS-relevant design metric is the eye diagram correction produced by the circuit. In typical AMS applications, the eye diagram at the input to the equalizer (*i.e.*, the channel output) has a very small or even non-existent eye opening (Fig. 5.16(a)). The equalizer offsets some of the ISI produced by the channel, which can considerably widen the eye opening; for example, Fig. 5.16(b) shows the eye diagram produced by a small-signal SPICE simulation of the above system, using SpiceOPUS [52]. Parts (c) and (d) of Fig. 5.16 depict the eye diagrams produced by applying 6-bit and 8-bit ABCD-L to the above "channel plus equalizer" system, overlaid on top of the (blue) SPICE eye diagram. As the figures show, the eye diagrams obtained from ABCD-L's Boolean mod-

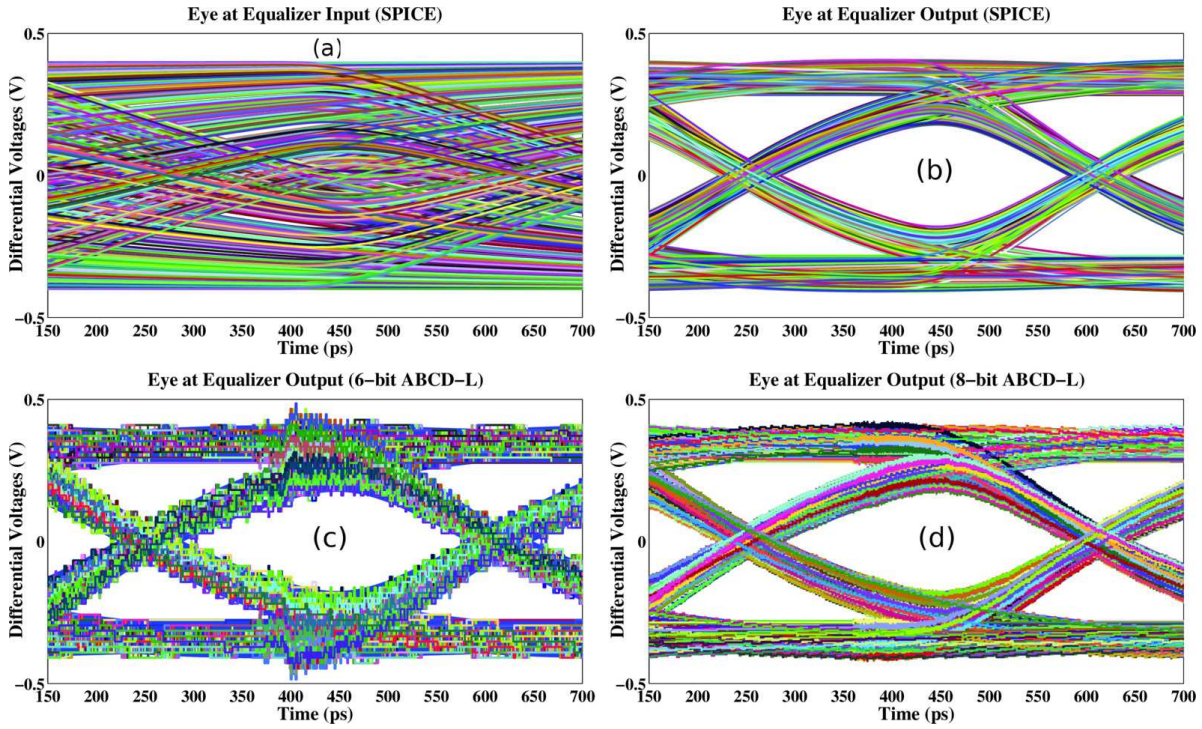


Figure 5.16: ABCD-L accurately reproduces the entire shape of the eye diagram at the equalizer's output.

els are able to accurately reproduce the eye diagram correction achieved by the equalizer. Thus, we have demonstrated that ABCD-L is a viable technique to Booleanize the continuous dynamics of AMS systems.

5.6 Example: Booleanizing a power-grid (ABCD-L + MOR)

Next, we combine ABCD-L with Arnoldi iteration based model order reduction (MOR) to efficiently produce accurate Boolean models of a real-world power grid network (with 25849 nodes) obtained from a benchmark set made available by IBM.

As described in Section 5.2, ABCD-L requires eigendecomposition of LTI systems as part of the Booleanization process. However, as LTI system size increases, eigendecomposition can quickly become computationally infeasible. To address this problem, we suggest an approach involving Model Order Reduction (MOR). The idea is to first obtain a Reduced Order Model (ROM) of the original LTI system; many well-established techniques are available for this, including explicit moment matching methods such as Asymptotic Waveform Evaluation (AWE) [63], implicit Krylov subspace methods such as congruent transformation [64], Padé

approximation via the Lanczos method [65], guaranteed-stable methods based on Arnoldi iteration [66,67], *etc.*. The next step is to apply ABCD-L to the ROM, which is typically much smaller than the original LTI system, and therefore not a barrier for eigendecomposition.

We now apply the above approach to a real-world power grid network, obtained from a benchmark set made available by IBM [68, 69]. This LTI network has 25849 nodes, making eigen-analysis slow and impractical. Therefore, we first carry out Arnoldi iteration [57] based MOR to reduce this system to a more manageable size. Indeed, as we show below, a ROM of size ∼20 suffices to capture the dynamics of this system for most frequencies of interest. We then Booleanize the ROM using ABCD-L, and show that the resulting Boolean model is able to accurately reproduce the behaviour of the original system, including such attributes as the power grid's voltage swings in the ground plane and in the V_{DD} plane.

5.6.1 Arnoldi ROMs for the power grid

Given an LTI system L_{orig} of size n, and a desired ROM size p (where $p \ll n$), the method of Arnoldi iteration produces an LTI system L_{ROM} of size p, that approximates the behaviour of the original system L_{orig}. The key idea behind Arnoldi MOR is to *match moments*, *i.e.*, the reduced order model L_{ROM} is constructed in such a way that the first p moments of its transfer function are identical to those of the original system L_{orig}. In addition to being fast, Arnoldi iteration has favorable numerical properties (in the context of fixed-precision computation), as opposed to other techniques like Padé approximation [40, 57].

To determine a suitable ROM size p for the IBM power grid, Fig. 5.17 compares the LTI frequency-domain transfer function (both magnitude and phase) of the original power grid against those of several different Arnoldi ROMs, corresponding to different sizes p, over a wide frequency range (1 Megahertz to 1000 Terahertz). The black waveform (with black square markers) represents the original system's transfer function, while the '*' marked color waveforms correspond to the Arnoldi ROMs (with sizes as labelled in the figure). The figure clearly brings out the effectiveness of Arnoldi ROM for this network; for example, even though the original system is of size 25849, a ROM of size ∼20 suffices to accurately capture the system's behaviour for input excitations upto 100 GHz. Therefore, an Arnoldi ROM of size ∼20 is more than adequate for most typical power grid applications.[5]

5.6.2 ABCD-L + Arnoldi MOR applied to the power grid

We now apply ABCD-L to produce purely Boolean approximations of the ROMs generated above. As noted earlier, these ROMs are much smaller systems compared to the original

[5]We note that Fig. 5.17 depicts the transfer function for only one output node of the power grid. In reality, the transfer functions corresponding to all output nodes (both in the power plane and in the ground plane) must be examined before concluding that ROM size $p = 20$ is adequate. For the given network, we have confirmed that this is indeed the case; however, to avoid unnecessary repetition, we do not include all the transfer function plots here.

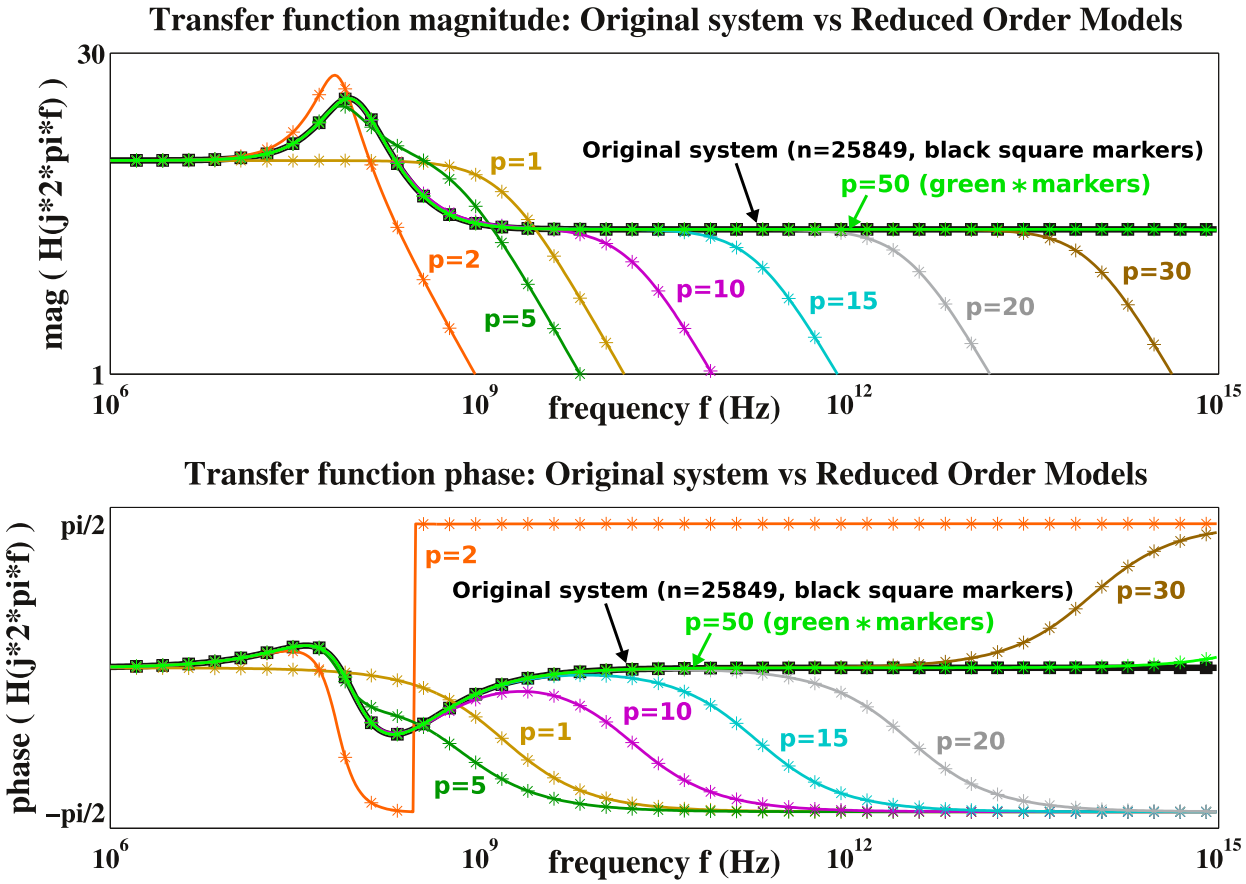


Figure 5.17: Arnoldi ROM applied to the IBM power grid network. As ROM size p increases, the reduced order models produced by Arnoldi iteration become better and better approximations of the original system.

power grid, and hence pose no difficulty for eigen-analysis (whose time complexity is cubic in the size of the given matrix).

Fig. 5.19 shows that the Boolean models produced by ABCD-L are able to accurately reproduce the transient dynamics of the Arnoldi ROMs generated for the IBM power grid. Parts (a), (b), (c), and (d) of the figure correspond to ROM sizes 5, 8, 10, and 20 respectively. In each case, the power grid was excited by the same input waveform $u(t)$ – a superposition of two damped sinusoidal excitations at 10 GHz, one positive and the other negative (see Fig. 5.18).

Each part of Fig. 5.19 depicts the following: (1) two output waveforms (one output from the power plane and one from the ground plane) produced by the original power grid (in dark gray), (2) the same outputs, as predicted by the respective reduced order model (in blue), and (3) the outputs as predicted by 5-bit ABCD-L applied to the corresponding ROM (in green).

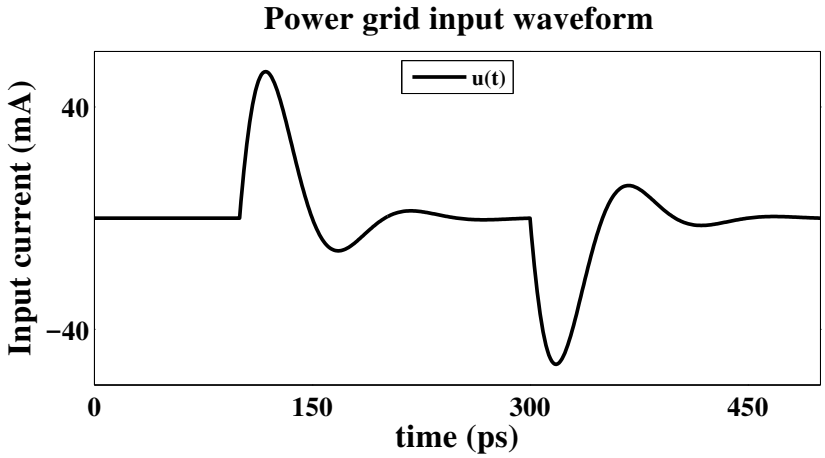


Figure 5.18: Input current waveform applied to the IBM power grid: a superposition of two damped sinusoidal excitations at 10 GHz, one positive and the other negative.

From the figure, it is clear that the Boolean models generated by ABCD-L always closely reproduce the corresponding ROM's response. Moreover, as the ROM size increases, this response in turn becomes a very good approximation to the response of the original power grid system, both in the power plane and in the ground plane.

Furthermore, as expected, the combination of ABCD-L with Arnoldi MOR resulted in significant computational savings over direct eigendecomposition of the original system. For instance, even with $p = 100$, the Arnoldi step took only about 10 minutes, on a 64-bit Linux machine equipped with a 3.2 GHz AMD® Phenom™ II X6 1090T processor and 16GB RAM. Moreover, once the Arnoldi iteration was completed, the time required for ABCD-L (including pre-processing, model generation, transient simulation, and post-processing) was well under a minute. This shows that ABCD-L, in conjunction with MOR, is indeed a viable technique that can be applied to accurately Booleanize even large LTI systems.

5.7 High-speed AMS simulation using ABCD-L

Having presented results pertaining to ABCD-L's accuracy, we now consider its computational efficiency. As outlined in Section 5.2, the Boolean models produced by ABCD-L lend themselves to efficient time-domain simulation carried out entirely in the discrete/logical domain, without having to solve differential equations. Even after taking into account the time taken to generate the ABCD-L models, ABCD-L can still be many times faster than conventional circuit simulation techniques like linear multi-step integration. This is illustrated in Fig. 5.20, which compares the total time taken by 4-bit, 5-bit, 6-bit, and 8-bit ABCD-L (total runtime includes pre-processing, model generation, simulation, and post-processing) against the time taken by Backward Euler integration, for simulating RC chains of various lengths on a long, randomly generated input bit pattern. As the figure shows, ABCD-L

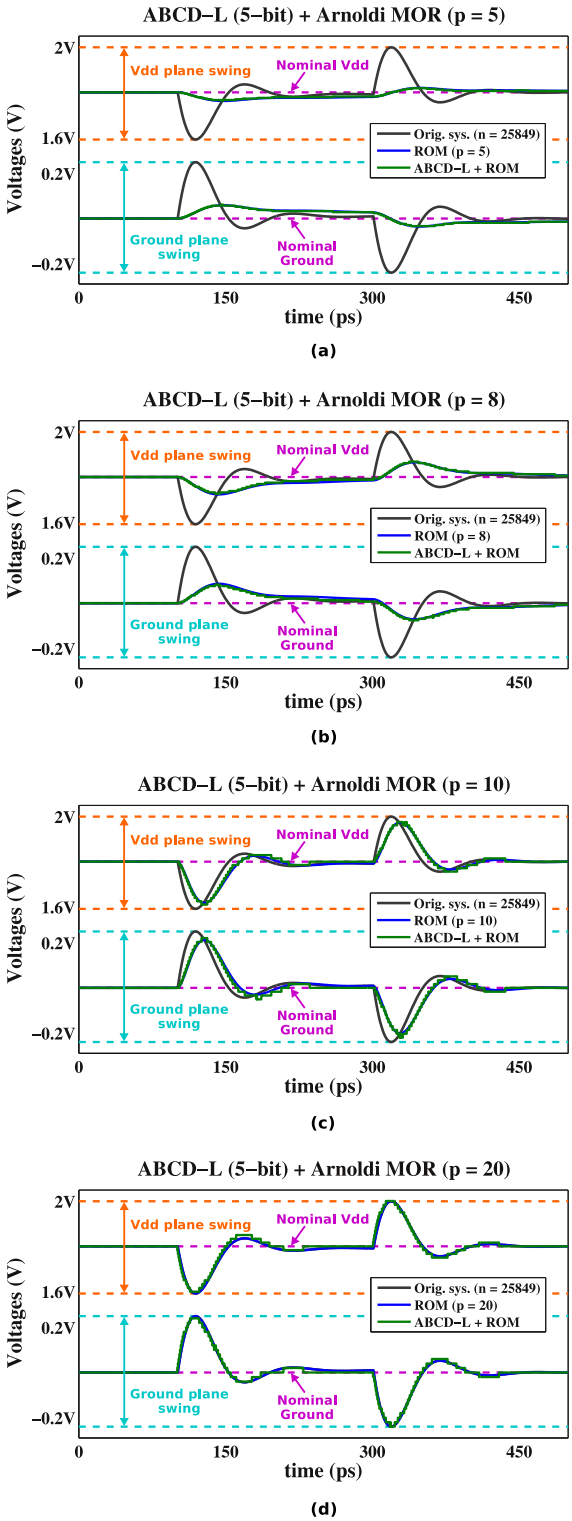


Figure 5.19: ABCD-L plus Arnoldi MOR applied to the IBM power grid, for different ROM sizes p. The plots depict the network's response to the input $u(t)$ of Fig. 5.18. In each case, it is seen that the Boolean model generated by ABCD-L closely approximates the ROM's response. And as ROM size increases, this response becomes a very good approximation to the response of the original power grid system.

does offer a significant speedup advantage. Moreover, as the LTI system size increases, this advantage becomes even more pronounced.[6] In addition, as discussed in Section 5.2 (and illustrated in the supplemental material), it is straightforward to integrate linear MOR techniques (*e.g.*, Arnoldi iteration [40,57]) into ABCD-L, which can further improve its runtime.

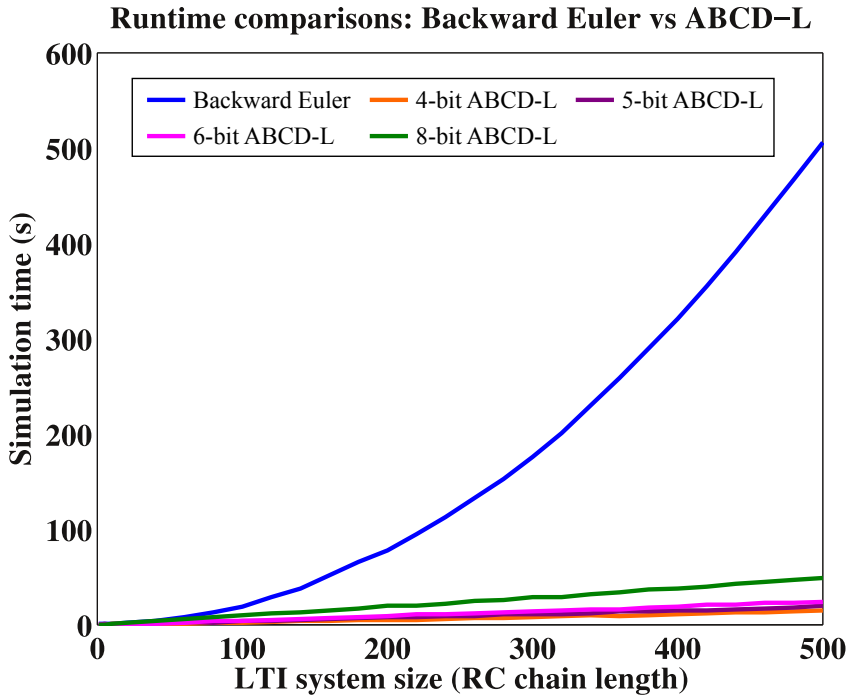


Figure 5.20: ABCD-L can offer considerable simulation speed-up over traditional circuit simulation techniques like Backward Euler integration. The figure illustrates this for RC chains of varying length, and for 4-bit, 5-bit, 6-bit, and 8-bit ABCD-L.

5.8 Summary

In this chapter, we have developed and demonstrated ABCD-L, a technique that automatically produces Boolean approximations of continuous LTI systems, to any desired level of accuracy, featuring a good balance between scalability and modelling accuracy. We have applied ABCD-L to representative LTI systems such as RC/RLGC chains, where it captures important analog effects like inter-symbol interference, ringing, *etc.*. We have also demonstrated ABCD-L on a small-signal linearized "channel plus equalizer" circuit, where it is able to reproduce key design-relevant AMS metrics, including the eye diagram correction

[6] All the ABCD-L simulations of Fig. 5.20 have been carried out in C++, on a system equipped with a 6-core 3.2 GHz AMD® Phenom™ II X6 1090T processor, and with a total of 16GB (shared) memory.

achieved by the circuit. Furthermore, we have demonstrated, using as example an on-chip power grid circuit made available by IBM, how ABCD-L can be used in conjunction with LTI MOR techniques to Booleanize large LTI systems (with tens of thousands of nodes) accurately. Also, we have shown that ABCD-L-generated models can offer significant simulation speed-up over conventional circuit simulation techniques like linear multi-step integration.

Chapter 6

ABCD-NL: Automated Booleanization of Non-Linear Systems

This chapter focusses on ABCD-NL, the last of three automated Booleanization techniques we have developed as part of the ABCD suite.

ABCD-NL is the most general and most powerful Booleanization technique that we have developed. Unlike DAE2FSM (which works well for "digitalish" systems but tends to fail when applied to even moderately complex analog circuits), and unlike ABCD-L (which works only for LTI systems), ABCD-NL works well for accurately Booleanizing a large class of genuinely non-linear analog and mixed-signal systems.

As with DAE2FSM and ABCD-L, ABCD-NL starts with a continuous domain dynamical system description of the given AMS system (for example, a SPICE netlist or a system of differential-algebraic equations). Given such a system description, ABCD-NL produces a purely Boolean model (*e.g.*, a sequential AIG, an FSM, or a BDD plus some flip-flops, *etc.*) that captures the I/O behaviour of the given system, with a high degree of accuracy, without making any *a priori* modelling simplifications, and often maintaining good scalability.

As with DAE2FSM and ABCD-L, the Boolean models produced by ABCD-NL can be used for high-speed simulation and formal verification of AMS designs by leveraging existing tools developed for both Boolean and hybrid system analysis (*e.g.*, ABC [6]).

In this chapter, we first motivate the need for a general-purpose non-linear Booleanization tool such as ABCD-NL. Then, we delve into the core techniques and concepts underlying ABCD-NL. Then, we demonstrate the power of ABCD-NL by applying it to a variety of non-linear AMS circuits modelled at the SPICE-level, including data converters, charge pumps, delay lines, comparators, non-linear signalling/communication sub-systems, *etc.*. In each case, we show that the Boolean models generated by ABCD-NL are able to match SPICE-level simulations with good accuracy, without unduly sacrificing scalability. Finally, we discuss the limitations of ABCD-NL.

For a more succinct treatment of the material in this chapter, we refer the reader to our paper on ABCD-NL [70].

6.1 The need for ABCD-NL

As described in Chapter 1, AMS systems are becoming increasingly important in chip design. In particular, non-linear AMS components like charge pumps, ADCs, DACs, PLLs, *etc.*, have come to play an integral rôle in AMS design. Furthermore, even the AMS components whose *intended* functionality is linear (*e.g.*, amplifiers, filters, *etc.*) tend to exhibit strongly non-linear behaviour outside a narrow range of operating conditions. This is especially true when analog/RF components are implemented using deep sub-micron nanoscale transistors with small feature sizes and limited driving strengths. Therefore, such non-linear blocks can easily become bottlenecks limiting system-level performance [3]. Also, non-linear AMS components have always been difficult for designers to understand and analyze, since the mathematics of non-linear dynamical systems is not as well-developed or as well-understood as the mathematics of LTI systems. For example, with non-linear systems, the notions of "transfer function" and "eigen-analysis" (which form the basis of our understanding of linear systems) no longer have any meaning. Not surprisingly, this difficulty in understanding non-linear systems often translates to a greater incidence of design bugs (and associated designer time and debugging costs) in such components and sub-systems.

As we discussed in Section 1.2.5, an important challenge for high-speed AMS simulation and AMS verification is the *accurate modelling* of AMS components; because these components can introduce design flaws/loss of performance in a variety of subtle and non-obvious ways, it is important to model their behaviour at or near SPICE-level accuracy, while still retaining scalability and amenability to analysis.

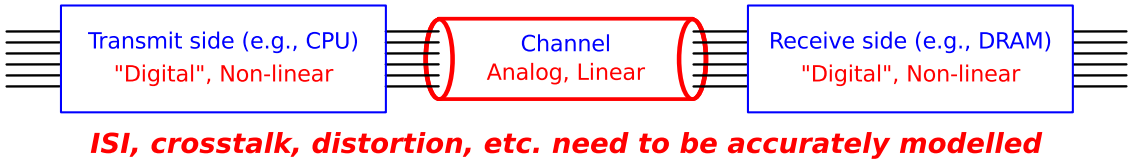


Figure 6.1: Schematic of a typical AMS signalling/communication sub-system that arises in signal integrity analysis.

For example, Fig. 6.1 depicts an AMS system that frequently arises in Signal Integrity (SI) applications. The system consists of digital components on the transmit side (*e.g.*, a CPU), whose outputs enter an analog channel. The channel introduces inter-symbol interference (ISI), crosstalk, *etc.*. The other end of the channel (the receive side) has more digital components (*e.g.*, a DRAM/memory controller). A key figure of merit of this system is its throughput, *i.e.*, the maximum bitrate that can be reliably sustained. Guaranteeing the system's throughput is a non-trivial AMS verification problem, and to issue a meaningful guarantee, it is often necessary to be able to model this system at or near SPICE-level accuracy. Indeed, very often, the throughput of these systems is checked only by carrying out a large number of time-consuming SPICE-level simulations, because other methods (such as hybrid system based verification methodologies) are often unable to achieve the degree of

modelling accuracy that AMS designers would be comfortable with. We suggest Booleanization, via the techniques described in this chapter, as an alternative.

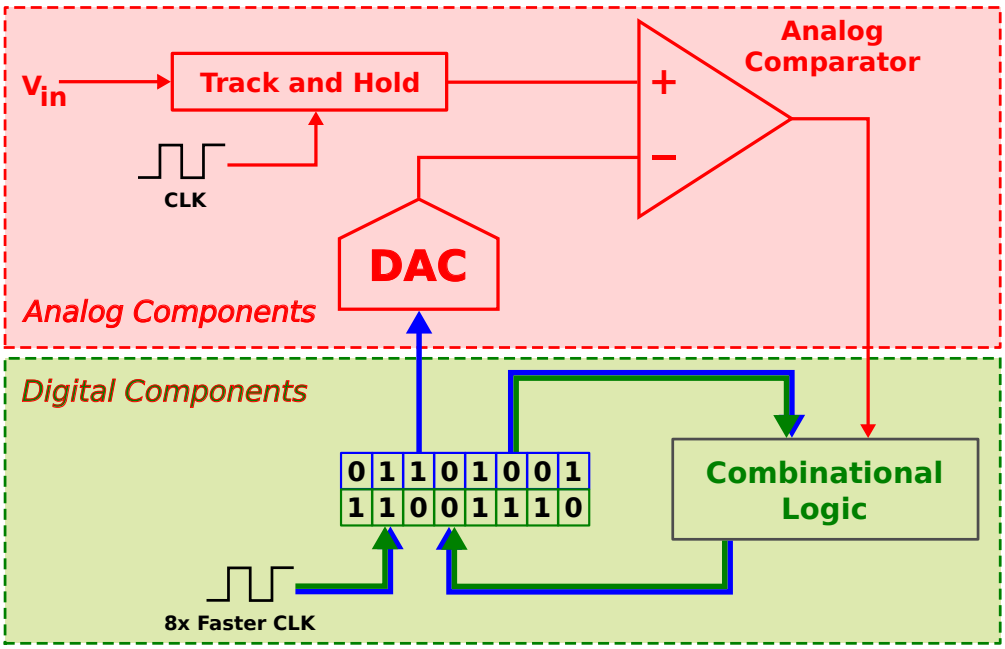


Figure 6.2: Schematic of a successive approximation A/D converter (SAR-ADC).

To take another example, Fig. 6.2 depicts a successive approximation A/D converter (or SAR-ADC), a system that we described earlier in Section 1.3.1. As indicated in the figure, this system contains a number of components, both analog and digital. For a detailed description of how these components work together to realize the ADC's functionality, please see Section 1.3.1. In practice, this ADC's performance (*i.e.*, its speed, power consumption, *etc.*) is frequently limited by the analog components (*e.g.*, the speed of the DAC, the sensitivity/bandwidth of the comparator, *etc.*). Therefore, in order to verify the correctness and performance of the SAR-ADC (or that of a larger system in which the SAR-ADC is a component), we suggest Booleanizing (at least) the analog components of the SAR-ADC using the techniques described in this chapter, in order to model their SPICE-level behaviour with good accuracy, and in a way that is amenable to formal verification.

The Booleanization techniques presented thus far in this dissertation (DAE2FSM and ABCD-L) are both unequal to the task of accurately Booleanizing genuinely analog and genuinely non-linear AMS designs, such as the signalling system of Fig. 6.1, or the SAR-ADC of Fig. 6.2. DAE2FSM is intended for the Booleanization of "digitalish" designs, and is often out of its depth when confronted with genuinely analog and/or mixed-signal designs (Section 4.8). ABCD-L, on the other hand, only applies to LTI systems; so it is cannot be used to Booleanize strongly non-linear AMS systems such as the ones in the examples above.

Therefore, there is a need to develop a new technique for Booleanizing strongly non-linear AMS designs, which this chapter aims to address.

We note that it is easy to come up with naïve Booleanization procedures for non-linear AMS designs based on discretizing the system state space and adding transitions between states using, for example, simple calculations based on forward/backward Euler integration (as described in Section 6.3.2). Such Booleanization procedures are easy to describe, and at first, they seem natural, intuitive, and "obviously correct". However, closer examination often reveals that such procedures produce Boolean models containing undesirable artifacts such as "fake fixed points". These are self-loops (or more generally, cycles) in the Boolean model that have no basis in reality; they cause the Boolean model to predict that the underlying system would settle into a stable state (or oscillate in a stable cycle) even when the original DAE features no such stable states/cycles. This can cause the Boolean model's behaviour to differ both *substantially* (in a quantitative sense) and *materially* (in a qualitative sense), from SPICE-level predictions. Therefore, while Booleanizing AMS systems, care must be taken to ensure that the Booleanization procedure that is followed does not lead to such discrepancies. The techniques developed in this chapter produce Boolean models that are guaranteed, by construction, not to contain such artifacts.

In this chapter, we propose ABCD-NL,[1] a Booleanization technique that works for a large class of non-linear systems. Given a non-linear AMS system (*e.g.*, one that is expressed as a SPICE netlist or as a DAE), ABCD-NL produces a purely Boolean model (*e.g.*, as an FSM, an AIG, or a BDD, *etc.*) that captures the dynamics of the given system to near SPICE-level accuracy. And as with DAE2FSM and ABCD-L, the Boolean models produced by ABCD-NL are well-suited for use with cutting-edge formal verification/model checking engines such as ABC [6], or with existing hybrid system frameworks. And just like the models generated by DAE2FSM and ABCD-L, the Boolean models produced by ABCD-NL can also be simulated very efficiently at the logic level (without needing to solve any differential equations), so ABCD-NL can be used as a much faster, almost-as-accurate, drop-in replacement for SPICE, in many applications involving non-linear AMS components and sub-systems.

But unfortunately, ABCD-NL, at this time, cannot handle all non-linear systems. In particular, for a circuit/system to be amenable to Booleanization via ABCD-NL, it must satisfy an important condition: a DC (constant) input to the given system should eventually result in a DC (constant) output. In practice, this condition is not hard to satisfy. Almost all non-linear systems of interest to AMS designers – for example, D/A and A/D converters, amplifiers and comparators, linear and non-linear filters, equalizers, switches and multiplexers, charge pumps, I/O links, *etc.* – satisfy this condition. The only practical, real-world exceptions we are aware of are oscillators. Please also see Section 6.9, where we discuss other situations where ABCD-NL may not apply.

The high-level ideas behind the inner workings of ABCD-NL closely match the steps outlined in Section 3.1. As described in this section, the first step involves discretizing the

[1] **A**ccurate **B**ooleanization of **C**ontinuous **D**ynamics - **N**on-**L**inear.

given circuit's inputs and outputs, plus perhaps a few internal voltages and/or currents as specified by the user. As with DAE2FSM and ABCD-L, ABCD-NL also offers fine-grained control over the discretization process. For example, the user can choose which circuit signals to discretize, how many bits/levels to use for each signal's discretization, how finely to discretize time (*i.e.*, the clock period/frequency of the resulting Boolean model), whether to use uniform or non-uniform discretization, *etc.*.

Once the discretization is done, the next step followed by ABCD-NL is to run a set of carefully selected SPICE simulations involving step-like input waveforms (for more details, please see Section 6.2). Finally, based on the output waveforms returned by these SPICE simulations, ABCD-NL constructs an FSM model for the given system. Usually, if a fine enough discretization is used, this Boolean model closely matches the SPICE-level dynamics of the given AMS circuit. As described in Section 3.4, it is possible to check the accuracy of the Boolean model generated by ABCD-NL simply by mapping its predictions back into the continuous domain (as a post-processing step) and comparing them againt the responses predicted by SPICE simulations for a range of different input waveforms.

The rest of this chapter is organized as follows: in the next section (Section 6.2), we describe in detail the core techniques and key concepts underlying ABCD-NL. Next, in Section 6.3, we illustrate these by presenting a detailed, step-by-step example where we use ABCD-NL to Booleanize a delay line (modelled behaviourally for simplicity and ease of exposition). In the next few sections, we apply ABCD-NL to Booleanize (starting from SPICE netlists) a number of practical, real-world, non-linear systems that are of interest to AMS designers. For example, in Section 6.4, we Booleanize a charge pump/filter system using ABCD-NL. We also apply ABCD-NL to the analog components that make up the SAR-ADC of Fig. 6.2, such as the D/A converter (Section 6.7) and the comparator (Section 6.8). We also apply ABCD-NL to Booleanize signalling systems of the form depicted in Fig. 6.1, as well as delay lines that play an important rôle in PLLs and DLLs. In all these cases, we show (using comparisons against SPICE) that ABCD-NL's Boolean models are able to faithfully reproduce the circuit's continuous-domain behaviour. Finally, in Section 6.9, we discuss the limitations of ABCD-NL.

6.2 Core techniques underlying ABCD-NL: Booleanizing non-linear analog systems by separating their DC and transient behaviours

In this section, we describe the key ideas behind ABCD-NL. As outlined in Section 6.1, ABCD-NL takes a SPICE netlist as input, and it produces as output a purely Boolean model (an FSM) that approximates the given circuit's dynamics.

As mentioned above, the Boolean model first represents the given circuit's inputs and outputs (as well as some internal voltages and currents in the circuit, if specified by the user) as discretized symbol sequences using finite numbers of bits for the discretization.

Indeed, when the Boolean model is simulated on an input sequence, the bits returned by it are interpreted by the user as discretized approximations to the circuit's output waveforms (following the discussion in Section 3.4). The goal, of course, is to preserve the behaviour of the original circuit as closely as possible in the Boolean model.

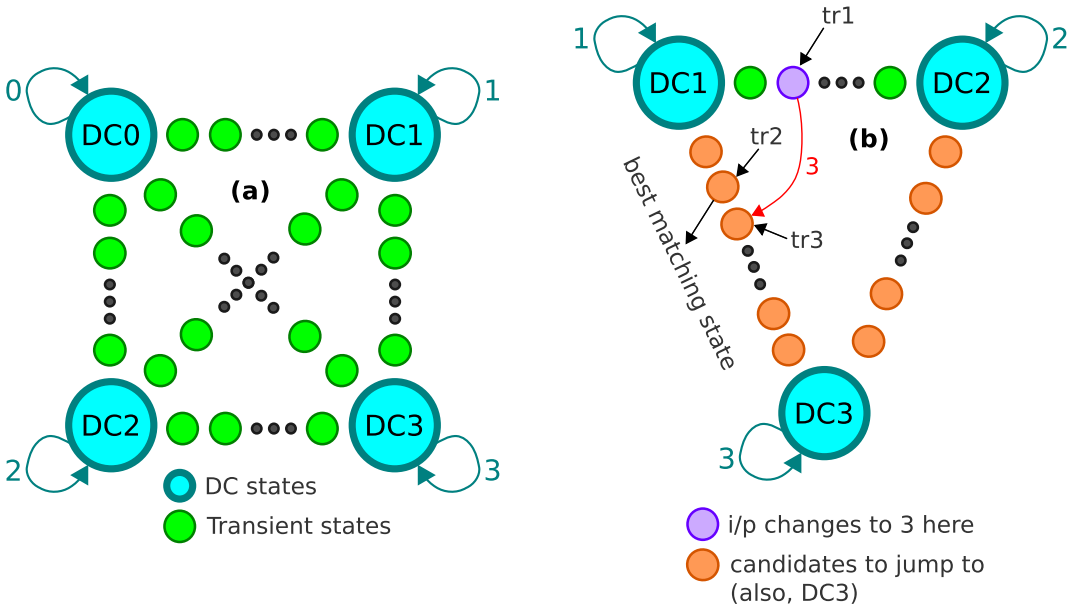


Figure 6.3: (a) Structure of the FSM model derived by ABCD-NL from SPICE simulations, and (b) ABCD-NL's method of exploiting continuity to jump from one transient FSM state to another.

Fig. 6.3 (a) depicts the structure of the FSM Boolean model produced by ABCD-NL. As shown in the figure, each discretized combination of the circuit's inputs is associated with a *DC state* in the FSM. These are the light blue states in Fig. 6.3 (a), that have self loops on them. These FSM states are known as DC states because they capture the DC behaviour of the given circuit.

As noted in Section 6.1, we have made the assumption that DC inputs to the circuit always eventually result in DC outputs. This makes it easy to characterise the DC behaviour of the circuit.

For example, let us say that the circuit has 2 inputs, and that each input is discretized using 3 bits (*i.e.*, 8 levels). Thus, there are 64 different DC input combinations possible. By our DC-to-DC assumption above, each of these 64 DC input combinations, when applied to the given circuit, will eventually result in the circuit converging to a *DC operating point*. Once the circuit converges to such a DC operating point, it is guaranteed to remain there as long as the input does not change (by virtue of the definition of a "DC operating point" in the continuous domain). Now we ask ourselves: what does this idea of "converging to a point and never changing" translate to in the discrete domain? The answer is an FSM state

with a self loop. Such an FSM state captures the notion that, once the system reaches this state, it will remain in this state forever, as long as the input symbol is unchanged (very much like the notion of DC operating points).

Returning to our example above, if we have 64 possible input combinations giving rise to 64 corresponding DC operating points, this DC behaviour can be modelled in our FSM using 64 DC FSM states, each with a self loop that makes sure of DC convergence. Each of these self loops, of course, is an arc in the FSM, and these arcs need to be annotated with input symbols and output symbols.[2] For example, let us say that our 64 input symbols are named 0 through 63. Then, the 20^{th} DC FSM state (corresponding to applying constant inputs that match the 20^{th} input combination) will contain a self-loop whose input symbol is 20. The output symbol on this self-loop will be the (discretized) output to which the circuit converges when the input symbol 20 is applied. In this way, we populate the FSM with 64 DC FSM states and 64 self-loops, with each self-loop being annotated with a relevant input symbol and an output symbol. These DC FSM states thus capture the given circuit's DC behaviour. Therefore, these DC FSM states are akin to DC operating points of the circuit.

Now we approach the question of how to capture the circuit's transient behaviour within our Boolean abstraction. For this, we consider the system's responses to step inputs (*i.e.*, sharp transitions from one DC value to another), and attempt to capture these responses as accurately as we can in the Boolean model. The motivation for considering step inputs is that, in typical real-world applications, AMS systems exhibit their most important and interesting dynamics when faced with inputs that are changing fast. Most well-designed AMS systems have enough "time to react", and hence behave as their designer intended, when their inputs are only changing slowly. On the other hand, when an AMS system's inputs are switching rapidly, the system is likely operating "closer to the edge", and scrambling to produce the correct outputs as quickly as possible. This is when many analog non-idealities, subtle features, and design bugs are likely to be brought to light. For example, in typical LTI systems, features like increased dispersion due to inter-symbol interference, overshoot/undershoot, ringing, *etc.*, are most often triggered by such "high-frequency" excitations. An ideal step input is the ultimate high-frequency excitation: it has infinite slope. So it is likely to reveal important and interesting information about the underlying system (even if the system is not LTI), which is why we consider it while building our Boolean model.

Specifically, to capture the circuit's transient behaviour, ABCD-NL introduces additional *transient FSM states* between every pair of DC states described above, using the system's step responses.

The logic for this is as follows. Let us consider an ideal step waveform that stays at one DC value (say, D_1) before a certain time t_0, transitions to another DC value (say, D_2) at time t_0, and remains at the new DC value forever. Now we ask the question: how will the given circuit respond to this step input?

By the DC-to-DC assumption above, the circuit will start out in the DC operating point corresponding to the input D_1 (and it will remain at this DC operating point for all times

[2] Recall, from Section 3.2, the definition of a Mealy machine.

$t < t_0$). Furthermore, as $t \to \infty$, the circuit will converge to a new DC operating point corresponding to the input D_2. In between $t = t_0$ and $t = \infty$, the circuit will most likely exhibit some sort of transient dynamics, which can be determined by running a SPICE simulation of the circuit with such a step waveform as input.

Algorithm 1: Converting an analog circuit into a purely Boolean ABCD-NL model

Inputs: SPICE netlist *cir*, Signal list *siglist*, Output signal *sigout*, FSM time step *tstepFSM*
Output: Purely Boolean FSM model *fsm* for the given circuit

```
1  fsm = new FSM()
2  DCInputs = enumerateDiscretizedCktInputs()
3  insertDCFSMStates(fsm, DCInputs)
4  for initInput in DCInputs:
5      initFSMState = encodeFSMState(initInput)
6      for finalInput ≠ initInput in DCInputs:
7          finalFSMState = encodeFSMState(finalInput)
8          cirInputWaveforms = generateStepFunctions(initInput, finalInput)
9          [ts, vs] = SPICESimulateTran(cir, cirInputWaveforms)
10         tSettle = estimateSettlingTime(ts, vs, siglist)
11         numTranFSMStates = ceil(tSettle ÷ tstepFSM) − 1
12         insertTranFSMStates(fsm, initFSMState, finalFSMState, numTranFSMStates)
13         annotateFSMArcs(fsm, initFSMState, finalFSMState, ts, vs, sigout)
14         annotateFSMStates(fsm, initFSMState, finalFSMState, ts, vs)
15 foreach transient FSM state trst in fsm:
16     foreach discrete input inp ≠ trst.finalInput in DCInputs:
           /* use state/output continuity to obtain new FSM arc */
17         nextst = estimateNextFSMState(fsm, trst, inp)
18         outp = estimateDiscreteOutputOnTransition(fsm, trst, nextst, inp)
19         insertFSMArc(trst, nextst, inp, outp)
20 return fsm
```

How do we capture this in the FSM? We do so by creating a *path* in the FSM that mimics the circuit's trajectory above. In other words, we create a path that starts at the DC FSM state corresponding to input D_1 and ends at the DC FSM state corresponding to input D_2. In between these end points, we introduce a number of additional FSM states that mimic the circuit's trajectory as predicted by SPICE.

For example, let us say that SPICE predicts a trajectory where the circuit's output starts at 0 Volts, crosses 0.25 Volts at time $t = 5$ns, and eventually reaches 0.50 Volts at time $t = 20$ns. If the Boolean model's clock period is 1ns, we can model this behaviour by introducing 19 transient states between the two DC state end points above. We can then

add 20 arcs that form a path beginning at the first DC state above, ending at the second DC state above, and including these 19 transient states in the middle. We can annotate the first 5 arcs with an output symbol corresponding to, say, an output of 0.125 Volts, and we can annotate the remaining 15 arcs with a different output symbol (corresponding to a circuit output of, say, 0.375 Volts). And we can annotate all these arcs with an input symbol corresponding to the DC value D_2. This will approximately capture the circuit's transient behaviour for the step input above.

In this way, we can conduct SPICE simulations of step inputs for every pair of DC FSM states in our Boolean model, and use the information to populate our FSM with a different transient path between every pair of DC FSM states. Thus, together, the DC FSM states and the transient FSM states capture the DC and transient dynamics of the given system to high accuracy.

Algorithm 1 formally describes the FSM construction procedure used by ABCD-NL at a high level. Line 3 creates the DC states described above. For every pair of such DC states, ABCD-NL performs a transient SPICE simulation (Line 9), the results of which are used to create the transient FSM states (Lines 10–12). Note that, if needed, the user can explicitly specify a list of important signals (*siglist*) in the given circuit, which the algorithm takes into account while creating the transient states. Further, the algorithm uses the SPICE simulations above to label each FSM arc (Line 13) with appropriate (discrete) input and output symbols, as discussed above.

By this time, all the states of the ABCD-NL FSM have been created, and all arcs have been specified for the DC states. However, not all arcs have been specified for the transient states. To fully specify the system, ABCD-NL uses an interpolation-based heuristic, known as a *jump heuristic*, as shown in Fig. 6.3 (b).

For example, suppose that the discretized version of the applied input switches from one Boolean-encoded value (say, 1), to another (say, 2). Corresponding to this, the FSM starts moving from state DC1 to DC2, as shown in Fig. 6.3 (b). However, before the FSM reaches DC2 (*i.e.*, when the FSM is in the state marked `tr1`), let us say the (discretized) input switches again, this time to 3. But with the arcs that exist in the FSM at this point, the behaviour of the FSM on this sudden input change is not specified, because the transient state `tr1` does not yet have an outbound arc corresponding to the input symbol 3. In other words, we need to add such an outbound arc to the state `tr1` so as to complete the FSM. The problem, of course, is to determine *where* this arc should lead to, as well as the *output symbol* on the arc. This is the problem that the *jump heuristic* is meant to solve.

We reason as follows: we know that the sudden change in the input symbol to 3 above will, if persisted, steer the continuous system towards state DC3. Therefore, our jump heuristic should nudge the FSM to move towards state DC3 as well – either by moving to state DC3 directly, or by moving to a state that will eventually converge to DC3 if the input symbol persists at 3. Also, we know that the underlying DAE system is continuous – so jumping directly to DC3 will probably not capture the continuous behaviour of the system very well. Instead, we need to find a state that is "as close as possible" to the current transient state `tr1`, which also has the property that it will eventually lead the FSM to DC3 if the input

persists at 3. For this, ABCD-NL takes advantage of the continuity of the underlying analog waveforms (Lines 17 to 19); it selects a transient FSM state that is, in some sense, "closest" to the current state `tr1`, along the paths DC1–DC3 and DC2–DC3 (this is possible because the ABCD-NL synthesis algorithm (Line 14) internally maintains an estimate of the *analog state* of the circuit at each *Boolean state* of the FSM). Having found such a "matching state" for `tr1`, we have the FSM do "whatever the matching state does" (in terms of the destination state and the output symbol) when the FSM is confronted with input 3 at state `tr1`. We then repeat this process for all transient states, and all input symbols for which they currently do not have outbound arcs. This completes the Boolean model generation, and the resulting FSM is returned, which, if necessary, can be transformed into an AIG or BDD using existing tools such as ABC [6]. Our implementation of ABCD-NL ensures that its output can be directly read in by tools such as ABC.

Finally, we note that the jump heuristic presented in Algorithm 1 is only an example. Other jump heuristics are also possible, as described in Section 6.3.6).

Algorithm 2: Simulating ABCD-NL's Boolean model, and post-processing to the analog domain

Inputs: Boolean model *fsm*, Ckt. inputs $u(t)$, FSM step *tstepFSM*, simulation interval $[t_0, t_f]$
Output: Simulation trace [*ts*, *vs*] of the circuit's output *sigout* over the interval $[t_0, t_f]$

```
1  ts = [ ], vs = [ ]
   /* start at the DC operating point for the input u at time t0 */
2  t_curr = t0
3  u_curr = discretizeInput(u(t_curr))
4  state_curr = encodeFSMState(u_curr)
5  while t_curr ≤ t_f:
       /* simulate one time point by looking up the FSM's state transition table */
6      transitionArc = fsm.nextStateArc(t_curr, u_curr)
       /* Look up the o/p on the transition arc, and post-process it from Boolean to analog */
7      output_curr = transitionArc.outputSymbol
8      analogOutput_curr = BooleanToAnalog(output_curr)
       /* record the output */
9      ts.append(t_curr)
10     vs.append(analogOutput_curr)
       /* update the simulation variables for the next time point */
11     t_curr += tstepFSM
12     u_curr = discretizeInput(u(t_curr))
13     state_curr = transitionArc.finalState
14 return [ts, vs]
```

Algorithm 2 formalises how the ABCD-NL Boolean model can be simulated in the time-domain, at the logic level. Each time step in this simulation is simply a table lookup (Line 6), so there are no differential equations or Newton-Raphson iterations involved. Thus, ABCD-NL based simulation is much faster than SPICE. For example, even though we have im-

plemented ABCD-NL in Python (a language not known for being fast), it was still 5x to 30x faster than HSPICE, for most examples presented in the next few sections. We believe that, if the code is re-written in C/C++, it would be quite straightforward to achieve ∼100x speedup over HSPICE.

6.3 Detailed example: Booleanizing a behavioural voltage controlled delay line

Having described the core techniques used by ABCD-NL, we now present a detailed example illustrating each of these concepts and algorithms. We believe that this example will provide clarity to the reader on every aspect of the Booleanization process followed by ABCD-NL.

6.3.1 The circuit: A behavioural voltage controlled delay line

The purpose of this section is to provide the reader with a detailed Booleanization example that illustrates all the core concepts and techniques described above. For the sake of clarity and expositional simplicity, we shall Booleanize a simplified behavioural model of an AMS circuit (*i.e.*, *not* a SPICE-level model) in this section. Please note that this is just for illustrative purposes – so that the reader develops a keen understanding and appreciation of the core concepts and techniques underlying ABCD-NL. In subsequent sections, we will Booleanize real-world AMS circuits described at the SPICE-level using advanced transistor models. But for this section, we will stick to a simple behavioural model.

The circuit that we will Booleanize is a voltage controlled delay line. A delay line works as follows: it has two inputs, V_{in} and V_{ctl}, and one output V_{out}. The output signal V_{out} is meant to be a time-delayed version of the input signal V_{in}, and the delay introduced between these two signals is controlled by the control voltage V_{ctl}. The higher the control voltage, the greater the delay between V_{in} and V_{out} (and vice-versa).

Fig. 6.4 shows a schematic for the delay line described above. The "delay" is achieved by a simple "RC filter" mechanism, but both the resistor and the capacitor in the filter are controlled by V_{ctl}. That is, the resistance R and the capacitance C are both (increasing) functions of the control voltage V_{ctl}, given by the equations $R = R_0 + \alpha V_{ctl}$ (with $R_0 = 300\Omega$ and $\alpha = 700\Omega/V$) and $C = C_0 + \beta V_{ctl}$ (with $C_0 = 0.3$pF and $\beta = 0.7$pF$/V$). So the system as a whole is non-linear (as a function of the inputs V_{in} and V_{ctl}), and is represented by an ODE of size 1, as follows:

$$\frac{d}{dt}V_{out} = \frac{V_{in} - V_{out}}{R(V_{ctl})C(V_{ctl})} \ . \tag{6.1}$$

Fig. 6.5 shows a quick SPICE simulation of the delay line above. The simulation consists of a train of 3 *pulses* applied to V_{in} – a 0.5ns pulse, followed by a 2ns pulse, followed by a 5ns pulse. This train of pulses is applied twice: once when the control voltage V_{ctl} is high (the left half of Fig. 6.5), and once when V_{ctl} is low (the right half of Fig. 6.5). As Fig. 6.5

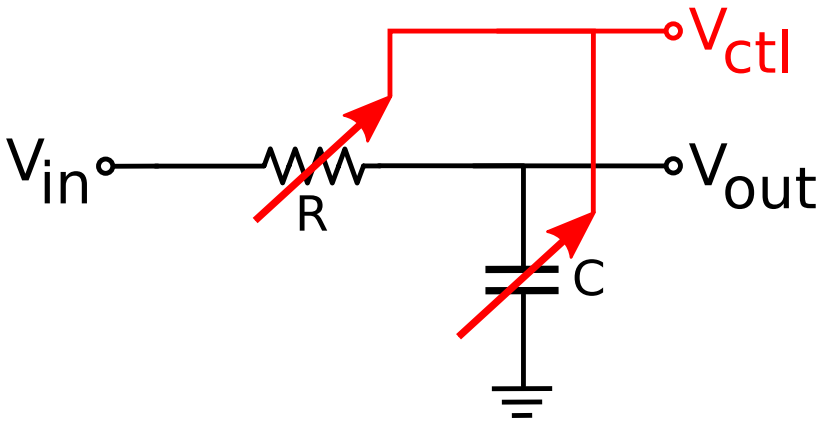


Figure 6.4: Schematic of a voltage controlled delay line modelled behaviourally.

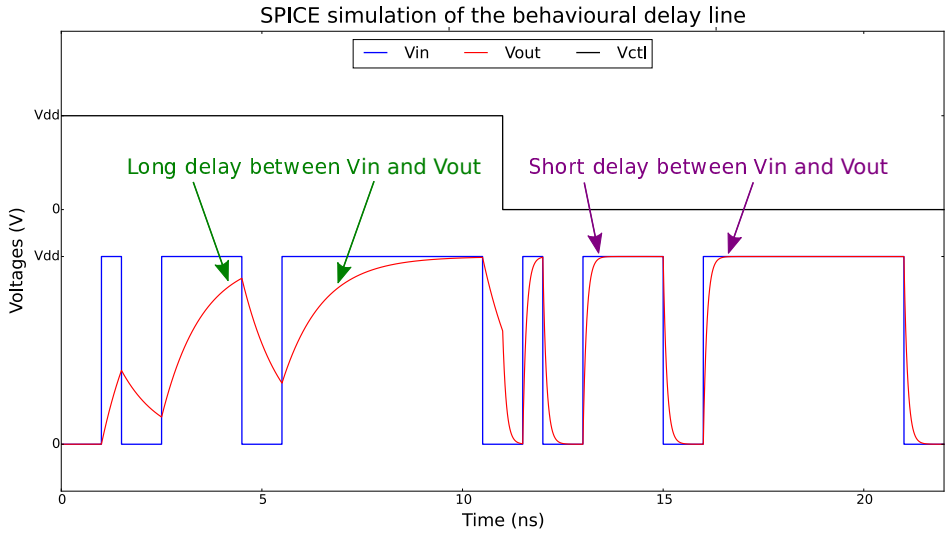


Figure 6.5: SPICE-simulation of the behavioural voltage controlled delay line.

clearly shows, when the control voltage is high, the "RC" delay between V_{in} and V_{out} is much larger, and vice-versa.

6.3.2 The fake fixed point problem in the context of this circuit

We now apply a naïve approach to Booleanizing the circuit of Fig. 6.4 – discretizing the state space of the circuit and encoding Backward-Euler based (continuous) state transitions via arcs in the discrete state space. However, this simplistic approach results in "fake fixed points", which we demonstrate below.

The (continuous) state space of the circuit above has just one variable, namely, V_{out}. Let us discretize this variable using, say, 8 levels of discretization: we denote these levels using the symbols 0 through 7, corresponding to the equally spaced quantized voltages 0 Volts, 1/7 Volts, 2/7 Volts, *etc.*, through 1 Volt. These correspond to 8 states (also denoted 0 through 7) in our FSM model. Thus, the FSM state space is simply a direct discretization of the system's continuous state space.

Further, let us discretize the inputs V_{in} and V_{ctl} using 1 bit each; that is, V_{in} and V_{ctl} can each be either 0 or 1, with 0 corresponding to 0 Volts and 1 corresponding to 1.0 Volt.

We now add arcs to our FSM by discretizing Backward Euler based state transitions in the original continuous system. Given the current (continuous) state $V_{out}(t_0)$ of the system at time t_0, and the inputs ($V_{in}(t_0+h)$ and $V_{ctl}(t_0+h)$) at the next time point (here, the next time point corresponds to 1 FSM clock period (denoted h) from the current time point t_0), the following Backward Euler equation [7] allows us to predict the (continuous) state of the system $v_{out}(t_0+h)$ at the next time point:

$$V_{out}(t_0+h) \approx \frac{\gamma V_{out}(t_0) + V_{in}(t_0+h)}{1+\gamma}, \text{ where}$$
$$\gamma = \frac{R(V_{ctl}(t_0+h))C(V_{ctl}(t_0+h))}{h} \ . \tag{6.2}$$

We now *discretize* the equation above as follows. For each discrete state S_0 from 0 through 7 of the FSM, and for each discrete FSM input symbol I_0 from 00 through 11 (where the first bit denotes V_{in} and the second bit denotes V_{ctl}), we first map S_0 and I_0 to continuous values for $V_{out}(t_0)$ and $(V_{in}(t_0+h), V_{ctl}(t_0+h))$ respectively. For example, if S_0 is 3, $V_{out}(t_0)$ is assigned the value 3/7 Volts, and so on. Similarly, if I_0 is 01, then $V_{in}(t_0+h)$ is assigned 0 Volts and $V_{ctl}(t_0+h)$ is assigned 1.0 Volt, and so on. Having assigned the continuous variables $V_{out}(t_0)$, ($V_{in}(t_0+h)$, and $V_{ctl}(t_0+h)$) for every possible discrete pair (S_0, I_0), we then use (6.2) to compute $V_{out}(t_0+h)$ in the continuous domain. We then *discretize* this computed value back into the 8-level discrete domain (resulting in a discrete state S_1 in the range 0 through 7) using a "nearest quantized level" approach. For example, if $V_{out}(t_0+h)$ (as predicted by (6.2)) happened to be 0.41 Volts, then this discretization would produce $S_1 = 3$, whose analog value $3/7 \approx 0.4286$ Volts happens to be the quantized level that is closest to 0.41 Volts out of the 8 available quantized levels. Thus, we can approximate (6.2)

```python
import itertools as it
from prettytable import PrettyTable as PT

def d2a(d, num_levels):
    return float(d/(num_levels-1))

def a2d(a, num_levels):
    levels = [float(i/(num_levels-1)) for i in range(num_levels)]
    distances = [abs(l-a) for l in levels]
    return distances.index(min(distances))

def R(vctl):
    return 300.0 + 700.0*vctl

def C(vctl):
    return 0.3e-12 + 0.7e-12*vctl

def BE(curr_vout, next_vin, next_vctl, tstep):
    RC_over_tstep = (R(next_vctl) * C(next_vctl))/tstep
    next_vout = (curr_vout * RC_over_tstep + next_vin)/(RC_over_tstep + 1.0)
    return next_vout

def BE_disc(curr_vout_disc, next_vin_disc, next_vctl_disc,
            tstep, num_levels_vout, num_levels_vctl):
    curr_vout = d2a(curr_vout_disc, num_levels_vout)
    next_vin = d2a(next_vin_disc, 2)
    next_vctl = d2a(next_vctl_disc, num_levels_vctl)
    next_vout_analog = BE(curr_vout, next_vin, next_vctl, tstep)
    next_vout_disc = a2d(next_vout_analog, num_levels_vout)
    return next_vout_disc

def main():

    num_levels_vctl, num_levels_vout, tstep = 2, 8, 10e-12

    vout_disc_fmt = '{:0' + str(len(str(num_levels_vout-1))) + 'd}'
    inp_disc_fmt = '{:01d}{:0' + str(len(str(num_levels_vctl-1))) + 'd}'

    col_titles = ['Current state', 'Next input', 'Next state', 'Remarks']
    alignments = 'rrrl'
    pt = PT(col_titles)
    for t, a in zip(col_titles, alignments):
        pt.align[t] = a

    combinations = it.product(range(num_levels_vout), range(2),
                              range(num_levels_vctl))

    for (curr_vout_disc, next_vin_disc, next_vctl_disc) in combinations:

        next_vout_disc = BE_disc(curr_vout_disc, next_vin_disc, next_vctl_disc,
                                 tstep, num_levels_vout, num_levels_vctl)

        curr_state = vout_disc_fmt.format(curr_vout_disc)
        next_inp = inp_disc_fmt.format(next_vin_disc, next_vctl_disc)
        next_state = vout_disc_fmt.format(next_vout_disc)

        remarks_list = []
        if next_vin_disc == 0 and curr_vout_disc == 0:
            remarks_list.append('Real fixed point')
        elif next_vin_disc == 1 and curr_vout_disc == num_levels_vout-1:
            remarks_list.append('Real fixed point')
        elif curr_vout_disc == next_vout_disc:
            remarks_list.append('Fake fixed point')

        if abs(curr_vout_disc - next_vout_disc) > 1:
            remarks_list.append('Discontinuous jump')

        remarks = ', '.join(remarks_list)

        pt.add_row([curr_state, next_inp, next_state, remarks])

    print(pt)

if __name__ == '__main__':
    main()

```

Figure 6.6: Python code that implements a naïve approach to Booleanizing the behavioural delay line circuit.

by adding an FSM arc from S_0 to S_1 on input symbol I_0 (the output symbol can simply be the next state of the FSM), for every possible combination of the discrete pair (S_0, I_0). The result is an FSM state transition table, which, at first glance, would seem to be a reasonable Boolean representation of the continuous system above.

Fig. 6.6 shows Python code that implements the naïve Booleanization approach above. This code, when run, produces a nicely formatted FSM state transition table; indeed, this is the code that was used to generate Table 6.1 and Table 6.2 below.

Table 6.1 shows the state transition table produced by the process above, using an FSM clock period of 10ps, a reasonable clock period to choose considering that the fastest dynamics exhibited by the given circuit has a time-constant of about 100ps. Unfortunately, as the table shows, this FSM simply ends up containing a lot of self-loops. This is because the *discretized* state of the circuit seldom changes within one FSM clock period: the associated continuous state of the circuit can indeed change during this time, but it tends to remain within a single discretization level, with the result that the change in the continuous state is not detectable at the discrete level. In other words, the continuous state of the system evolves so slowly relative to the FSM clock period that, in one FSM clock period, the change in the continuous state space is not enough to warrant a change in the discrete state space, resulting in an FSM self-loop. Such a self-loop in the FSM would imply that the system settles to a fixed point. However, in reality, the system is simply evolving slowly relative to the FSM clock period, and does *not* settle to a fixed point, leading to a big qualitative difference in behaviour between the original continuous system and the FSM produced above. We refer to these spurious self-loops as *fake fixed points*; they are highly undesirable artifacts that are brought about when continuous systems are Booleanized using naïve approaches such as the one followed above. The fake fixed points in Table 6.1 are highlighted in red.

Fig. 6.7 compares the predictions made by the FSM of Table 6.1 against SPICE, for randomly generated 2-level input waveforms V_{in} and V_{ctl}. From the figure, it is clear that the fake fixed points have a dramatic effect: indeed, the FSM stubbornly settles into a fake fixed point early in the simulation, and does not emerge from this fake fixed point until the end of the simulation, producing a waveform that is completely inaccurate relative to SPICE.

From the discussion above, it may seem tempting to simply increase the clock period of the FSM so as to eliminate fake fixed points. After all, if the change in the system's continuous state space within a single FSM clock period is not large enough to merit a change in the discrete state space, one might make the reasonable sounding argument that this problem can be avoided simply by increasing the FSM clock period, so that changes in the continuous state space become large enough to avoid creating fake fixed points. However, this is not a good idea, for several reasons. Firstly, this requires the FSM clock period to be chosen very carefully depending on the dynamics of the circuit being Booleanized; therefore, finding an appropriate FSM clock period (that completely eliminates all fake fixed points) is likely to be a very difficult problem. Secondly, as one increases the FSM clock period, one makes the underlying discretization much coarser – this is akin to sampling a waveform at a much lower rate than the rate at which it is evolving. This leads to significant degradation in time-resolution, *e.g.*, one can lose the ability to resolve changes that occur 1ps apart from

Current state	Next input	Next state	Remarks
0	00	0	Real fixed point
	01	0	Real fixed point
	10	1	
	11	0	Fake fixed point
1	00	1	Fake fixed point
	01	1	Fake fixed point
	10	2	
	11	1	Fake fixed point
2	00	2	Fake fixed point
	01	2	Fake fixed point
	10	2	Fake fixed point
	11	2	Fake fixed point
3	00	3	Fake fixed point
	01	3	Fake fixed point
	10	3	Fake fixed point
	11	3	Fake fixed point
4	00	4	Fake fixed point
	01	4	Fake fixed point
	10	4	Fake fixed point
	11	4	Fake fixed point
5	00	5	Fake fixed point
	01	5	Fake fixed point
	10	5	Fake fixed point
	11	5	Fake fixed point
6	00	5	
	01	6	Fake fixed point
	10	6	Fake fixed point
	11	6	Fake fixed point
7	00	6	
	01	7	Fake fixed point
	10	7	Real fixed point
	11	7	Real fixed point

Table 6.1: State transition table for an FSM obtained by naïvely discretizing the Backward-Euler transition function of the behavioural delay line circuit shown in Fig. 6.4, with the FSM clock period set to 10ps. The transitions corresponding to fake fixed points are highlighted in red.

Current state	Next input	Next state	Remarks
0	00	0	Real fixed point
	01	0	Real fixed point
	10	3	Discontinuous jump
	11	0	Fake fixed point
1	00	1	Fake fixed point
	01	1	Fake fixed point
	10	4	Discontinuous jump
	11	1	Fake fixed point
2	00	1	
	01	2	Fake fixed point
	10	4	Discontinuous jump
	11	2	Fake fixed point
3	00	2	
	01	3	Fake fixed point
	10	5	Discontinuous jump
	11	3	Fake fixed point
4	00	2	Discontinuous jump
	01	4	Fake fixed point
	10	5	
	11	4	Fake fixed point
5	00	3	Discontinuous jump
	01	5	Fake fixed point
	10	6	
	11	5	Fake fixed point
6	00	3	Discontinuous jump
	01	6	Fake fixed point
	10	6	Fake fixed point
	11	6	Fake fixed point
7	00	4	Discontinuous jump
	01	7	Fake fixed point
	10	7	Real fixed point
	11	7	Real fixed point

Table 6.2: State transition table for an FSM obtained by naïvely discretizing the Backward-Euler transition function of the behavioural delay line circuit shown in Fig. 6.4, with the FSM clock period set to 75ps. The transitions corresponding to fake fixed points are highlighted in red, while those corresponding to discontinuous jumps in the discretized state space are highlighted in blue.

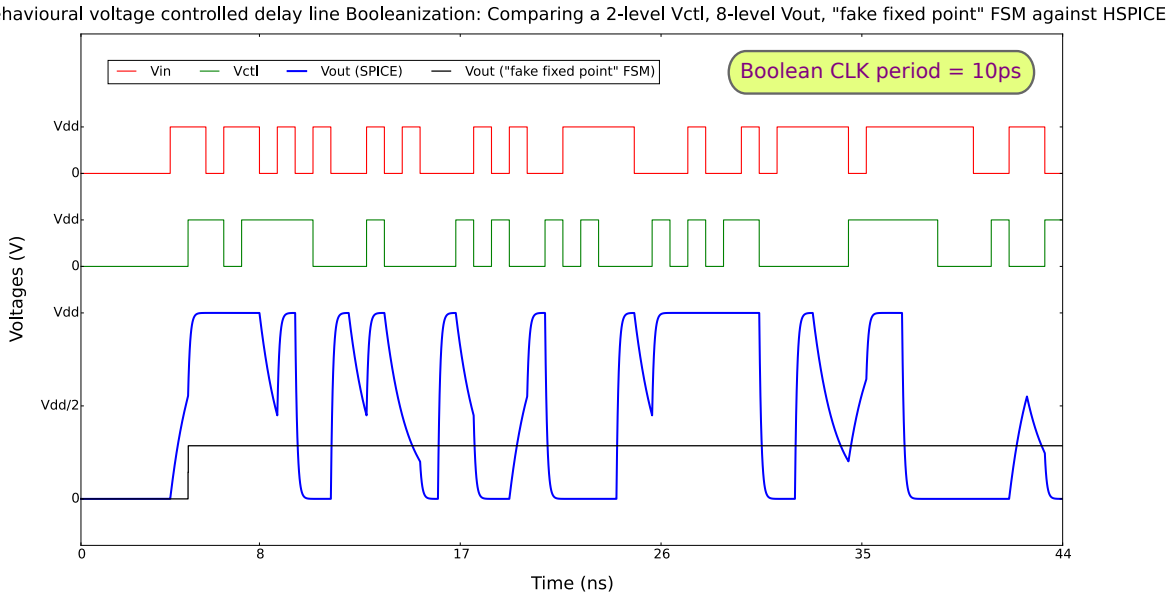


Figure 6.7: Comparing the predictions made by the FSM of Table 6.1 against SPICE.

changes that occur 5ps apart. Finally, when one increases the clock period, one also increases the risk of creating "non-smooth state transitions". For example, the discrete state could change in a single transition from the level 1 to the level 4, without ever going through levels 2 and 3, which is clearly not physically possible and which violates the continuity of the underlying AMS system. Thus, while some fake fixed points may indeed be eliminated as the FSM clock period is increased, such "discontinuous jumps" may be introduced into the FSM before all fake fixed points are eliminated.

For example, Table 6.2 depicts the state transition table produced using the naïve Backward Euler discretization approach above, but using 75ps for the FSM clock period instead of the 10ps clock that was used to produce the FSM in Table 6.1. As the table shows, this FSM indeed has fewer fake fixed points (highlighted in red); however, considering that the fastest time constants in the circuit are of the order of 100ps, this FSM does not have good time-resolution at a clock period of 75ps: one would like the clock period to be much lower. Also, even at 75ps, a number of spurious discontinuous jumps are introduced into the FSM (highlighted in blue). These correspond to changes that take place in the circuit that span more than one discrete level (*i.e.*, changes that cross multiple discrete thresholds) within a single FSM clock period.

Fig. 6.8 compares the predictions made by the FSM of Table 6.2 against SPICE, for randomly generated 2-level input waveforms V_{in} and V_{ctl}. From the figure, it is seen that the FSM often mistakenly "settles" into fake fixed points, and often emerges from such fake fixed points only to undergo "discontinuous jump" transitions to keep up with the continuous system. Qualitatively, the accuracy of this FSM is indeed better than that of the FSM shown in Table 6.1. However, in addition to the problem of unphysical discontinuous jumps, the

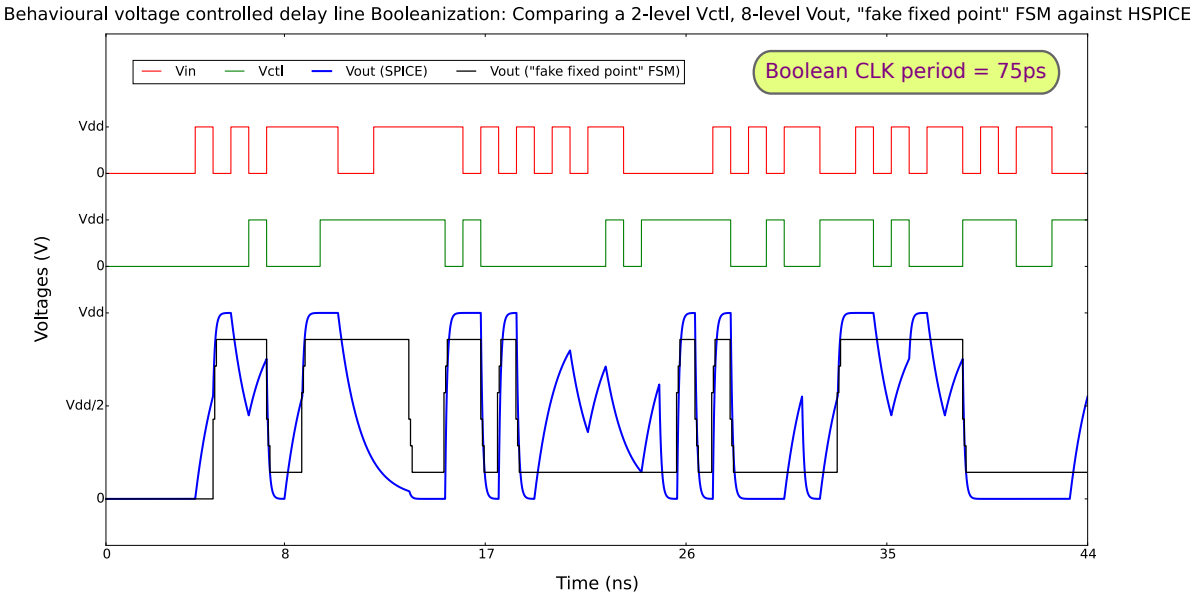


Figure 6.8: Comparing the predictions made by the FSM of Table 6.2 against SPICE.

clock period of this FSM, at 75ps, is very slow and not well-suited for capturing the fastest dynamics of the given circuit, which have time constants of about 100ps.

Thus, naïve discretization of transitions between the continuous and discrete state spaces often does not produce good Boolean models. For accurate Booleanization, the underlying Boolean model needs to have a notion of "changes in the discrete state space that occur over the passage of time". In other words, the underlying Boolean model needs counters (or equivalently, FSM TRAN states) that keep track of how much time has passed since the previous discrete change, and whether or not it is time for a new change. This is well taken care of by the Booleanization techniques proposed in Section 6.2. Below, we apply these techniques to the behavioural delay line, and show that they perform much better than the naïve approaches above.

6.3.3 DC states of the ABCD-NL FSM

As described in Section 6.2 above, the first step in the construction of the ABCD-NL FSM is to identify the DC states of the FSM. To do so, we must first decide on a discretization of the signals in the ODE above.

Let us start with the simplest possible discretization of the circuit's inputs. Let us discretize the input V_{in} using 1 bit, and the input V_{ctl} using another bit. That is, the inputs V_{in} and V_{ctl} are each independently allowed to be either 0 Volts (corresponding to the discrete level 0) or 1.0 Volt (corresponding to the discrete level 1). Furthermore, let us say that we wish to discretize the circuit's output V_{out} into 4 levels of discretization (denoted 0 through

DC FSM state	FSM input symbol	V_{in} (V)	V_{ctl} (V)	V_{out} (V)	FSM output symbol
DC_00	00	0.0	0.0	0.0	0
DC_01	01	0.0	1.0	0.0	0
DC_10	10	1.0	0.0	1.0	3
DC_11	11	1.0	1.0	1.0	3

Table 6.3: Continuous numerical values associated with the DC FSM states shown in Fig. 6.9.

3) between 0 Volts and 1 Volt. Let us also say that our Boolean FSM is clocked at about 10ps, a conservative clock that captures most of the dynamics exhibited by the circuit above.

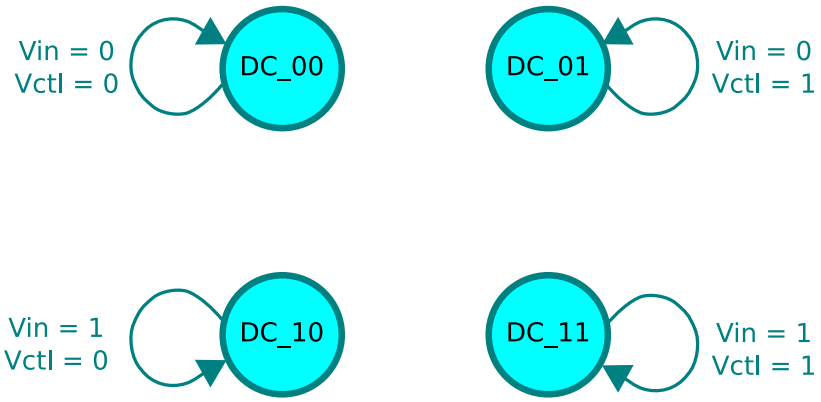


Figure 6.9: Capturing the DC operating points of the behavioural voltage controlled delay line using DC FSM states.

With the above discretization, there are now 4 distinct combinations of circuit inputs, and these 4 inputs each correspond to a unique DC operating point, thereby giving rise to 4 DC operating points. In the Boolean model generated by ABCD-NL, each of these DC operating points is represented by a DC FSM state, and therefore our FSM will have 4 DC states (as shown in Fig. 6.9). The continuous numerical values associated with these DC states are specified in Table 6.3.

6.3.4 TRAN states of the ABCD-NL FSM

Having fixed the DC states of the FSM, we are now ready to complete the state space of the FSM by adding a set of TRAN FSM states between every pair of DC FSM states, as described in Section 6.2 above. To add these TRAN states, we carry out some SPICE simulations of the circuit, using step waveforms as inputs (Section 6.2 explains why), as shown in Fig. 6.10.

In this FSM, there are 4 DC states. Therefore, we can enumerate $4^2 = 16$ possible TRAN paths, since each TRAN path corresponds to starting at one of the 4 DC states and

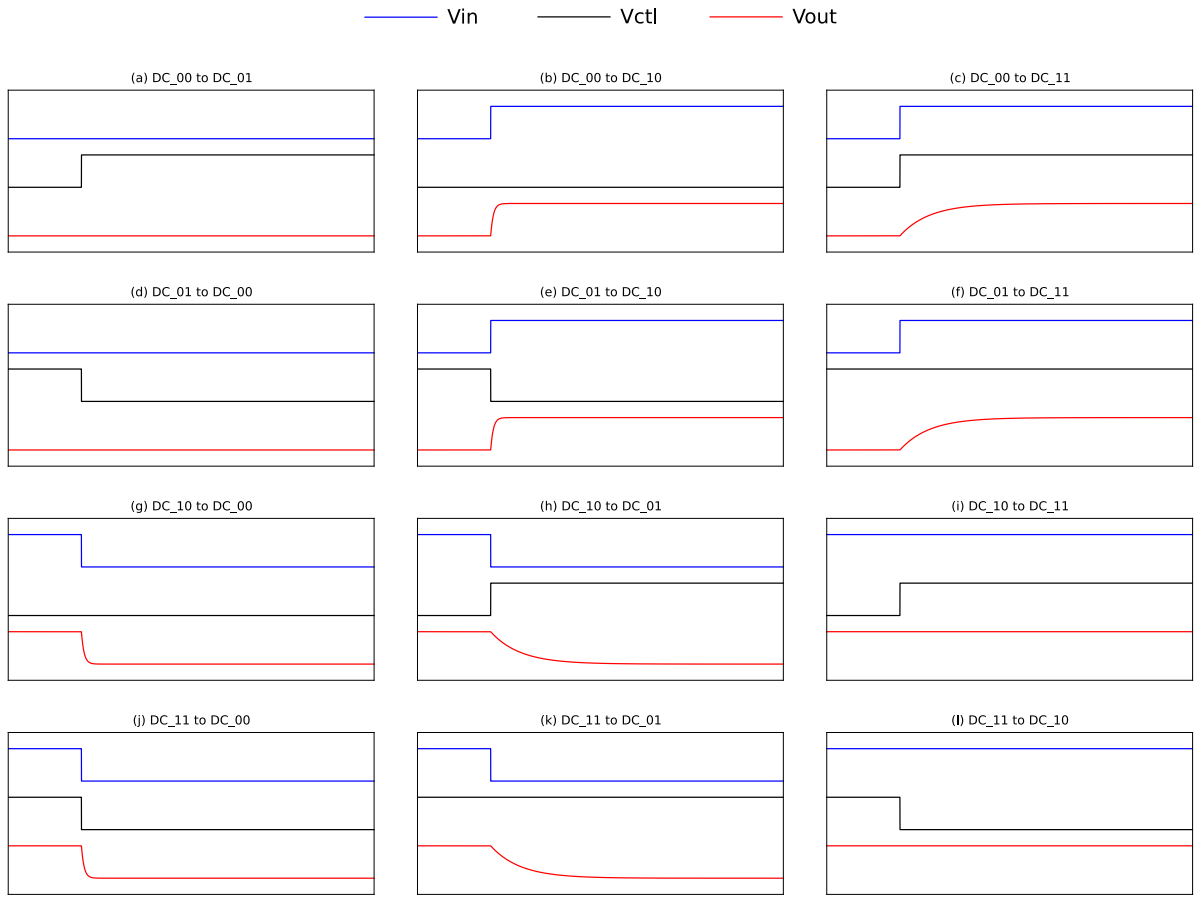


Figure 6.10: Running a set of transient SPICE simulations on step waveform inputs to learn the TRAN FSM states during the behavioural voltage controlled delay line's Booleanization. The titles of the subplots indicate the TRAN path that is learned from each simulation. The X-axis in each subplot represents time, while the Y-axes represent voltages.

then ending at one of the 4 DC states. However, we do not need to consider TRAN paths between a DC state and itself,[3] so this eliminates 4 of the 16 TRAN paths. Thus, we need to characterize 12 different TRAN paths, by carrying out 12 corresponding SPICE simulations. These are the 12 subplots in Fig. 6.10, where the title of each subplot indicates the start DC state and end DC state of the corresponding TRAN path.

For example, let us consider the TRAN path that starts at `DC_00` and ends at `DC_10` (with reference to Fig. 6.9). The start DC state corresponds to the circuit inputs V_{in} and V_{ctl} both being 0 Volts, while the end DC state corresponds to V_{in} being 1 Volt and V_{ctl} being 0 Volts. Thus, the SPICE simulation we need to run to determine the TRAN states along this path

[3]This is because, a TRAN path between a DC state and itself would correspond to a "step" input with identical values before and after the "step". In effect, this is a DC input that has already been accounted for by the self-loop on the DC state.

would correspond to a step input where V_{in} switches from 0 Volts to 1 Volt, while V_{ctl} is held steady at 0 Volts. This SPICE simulation is shown in Fig. 6.10 (b).

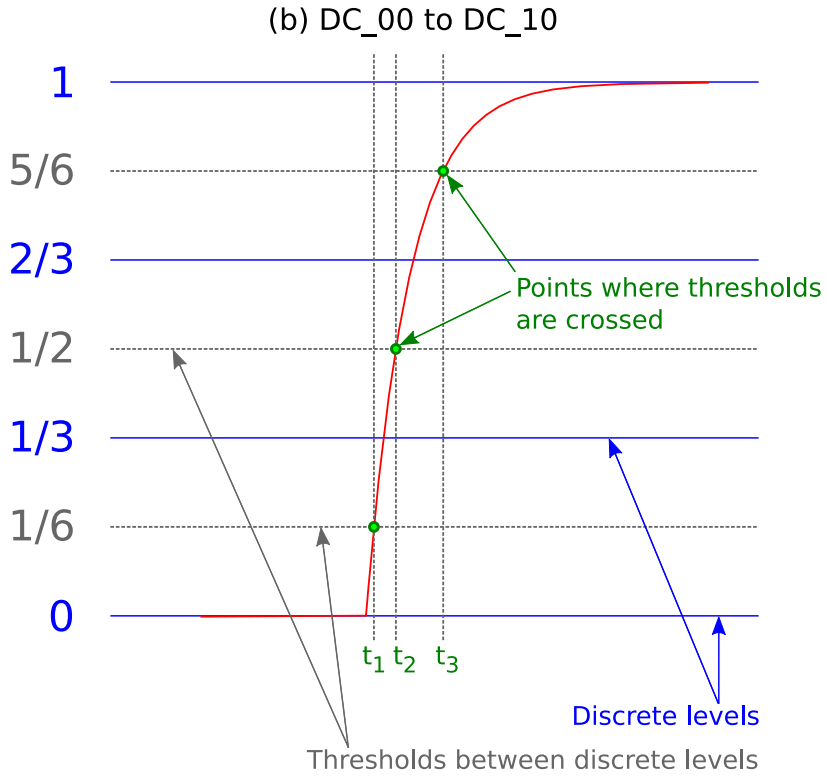


Figure 6.11: Recording the thresholds at which the output waveform for the DC_00 to DC_10 step input crosses from one discrete level to another.

To learn the TRAN path that starts at DC_00 and ends at DC_10, we now look at the output waveform produced by the circuit for the step input above. This output waveform is shown in red in Fig. 6.10 (b). As this subplot shows, the output waveform starts at 0 Volts and eventually converges to 1.0 Volt. In particular, since we are discretizing the circuit's output (uniformly) into 4 levels between 0 Volts and 1.0 Volt, these discrete levels fall at 0 Volts, 1/3 Volts, 2/3 Volts, and 1.0 Volt. Adopting a "quantize to the nearest discrete level" strategy, we are interested in the times at which the output waveform (which goes from 0 Volts all the way to 1.0 Volt) crosses the thresholds at 1/6 Volts, at 1/2 Volts, and at 5/6 Volts. This is illustrated in Fig. 6.11, where the time points that we are interested in are marked in green, and are labelled t_1 through t_3. The precise numerical values for these "threshold crossover times" are provided in Table 6.4.

Having found the times t_1, t_2, and t_3 from the SPICE simulation (Fig. 6.11), we can construct the TRAN path from the DC state DC_00 to the DC state DC_10 by calculating the number of states along the path that should produce the output 0, the number of states that should produce the output 1, *etc.*.

Time instant	Threshold crossed (V)	At time (ps)
t_1	1/6	16.9
t_2	1/2	62.9
t_3	5/6	161.7

Table 6.4: Numerical values for the time instants at which the thresholds shown in Fig. 6.11 are crossed by the behavioural delay line, when the input V_{in} instantaneously switches from 0.0V to 1.0V at time 0ps while the input V_{ctl} is held constant at 0.0V (corresponding to the TRAN path that starts at `DC_00` and ends at `DC_10`).

The number of states that should produce the output 0 is simply given by $t_1/10ps$ (rounded to the nearest integer). This is the amount of (continuous) time during which the system's output can reasonably be approximated by the discrete level 0, divided by the clock period of the Boolean FSM model. From Table 6.4, $t_1 = 16.9$ps, so 2 states along the `DC_00` $\rightarrow$ `DC_10` TRAN path should produce the output 0. One of these states is `DC_00` itself; the other is the first TRAN state along this path.

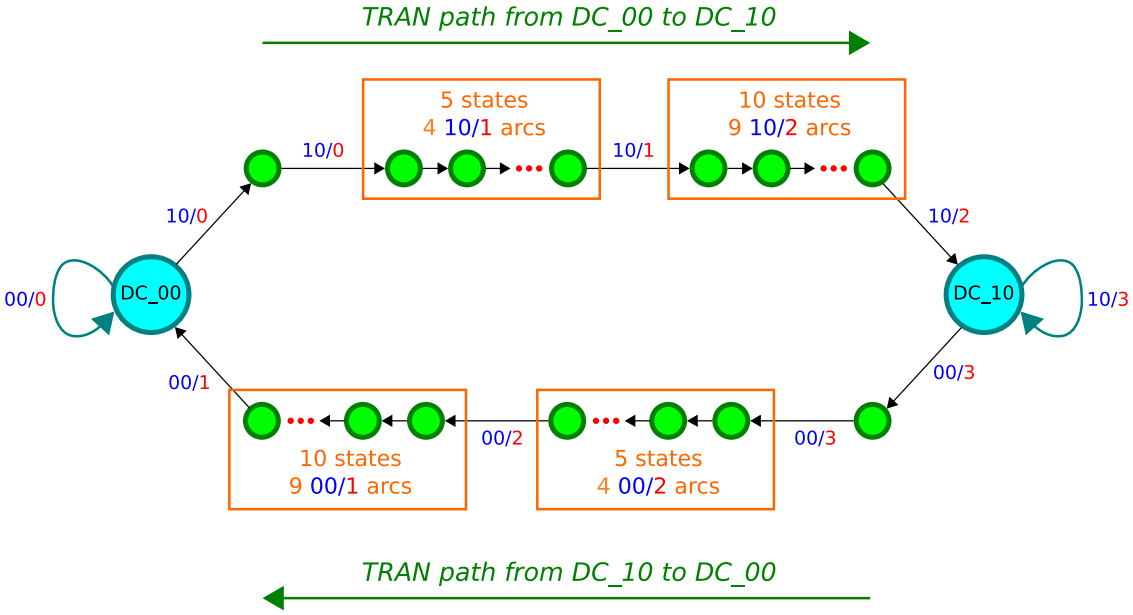


Figure 6.12: TRAN paths `DC_00` $\rightarrow$ `DC_10` and `DC_10` $\rightarrow$ `DC_00` in the ABCD-NL FSM for the behavioural delay line circuit.

Similarly, one can calculate the number of TRAN states that should produce each of the outputs from 1 through 3 (since the output is discretized into 4 levels). For example, the amount of time for which the circuit's output can be reasonably approximated by the discrete level 1 is $t_2 - t_1$, which from Table 6.4 is 46ps. So we should introduce 5 TRAN

states that each produce the output 1. Similarly, we should introduce 10 TRAN states that each produce the output 2, and the last of these arcs could end up at DC_10. Thus, the total number of TRAN states along the path is known, and so are the input/output symbols for each arc along the TRAN path. Fig. 6.12 shows the entire DC_00 → DC_10 TRAN path, as well as the reverse (DC_10 → DC_00) TRAN path.

The procedure above is repeated for every pair of distinct DC states in the FSM, which results in a partially complete Boolean model with both DC states and TRAN paths filled in. However, the FSM is not yet fully complete, *i.e.*, there still exist many input sequences for which the FSM does not have a corresponding path. The next sections cover the concept of *jump heuristics*, which are techniques that we use to fill these gaps in the FSM description.

6.3.5 The need for a jump heuristic

Let us now take a moment to describe why the FSM learned so far is not yet complete. One way to explore this is to ask the question: does every state in the FSM know how to handle every possible circuit input?

For example, consider the 4 DC states that we constructed above (Fig. 6.9). Do these DC states know how to handle every possible (discretized) input combination? The answer is yes, because, by our construction above, each DC state has an arc directed outwards for every possible (discretized) combination of the circuit's inputs. Given any such input, a DC state will either loop back to itself, or it will start moving along an appropriate TRAN path meant for that input. For example, in our FSM for the behavioural delay line, if the DC state DC_01 encounters the input 01, it will loop back to itself. If, on the other hand, it encounters the input 11, it will start transitioning along a TRAN path (constructed above), and this TRAN path will eventually culminate in the DC state DC_11.

However, the key question here is: what if, in the middle of such a transition, the input changes *again*? That is, let us assume that the FSM starts out in state DC_01. Now, let us say that the FSM receives the input 11, which stays at 11 for 5 FSM clock cycles. At the end of these 5 clock cycles, it is clear that the FSM would have reached the 5^{th} TRAN state along the TRAN path that starts at DC_01 and ends at DC_11 (assuming that there are at least 5 TRAN states along this path).

Now the FSM is in the 5^{th} TRAN state along this path. At this state, if the next input is again 11, there is no problem: the FSM simply advances along this TRAN path. However, what if the next input read is 10? In this case, the TRAN state does not know what to do, *i.e.*, the TRAN state has no arc directed outwards for such an input symbol. Thus, the FSM is incomplete. This is illustrated in Fig. 6.13.

In other words, every TRAN state constructed in the FSM so far "knows" how to handle only one input symbol. This means that, if a TRAN state encounters any input symbol other than the one it was set up for, the FSM would reach an impasse and there would be no way forward.

Therefore, the TRAN states in the FSM must be *augmented* with arcs for handling all possible input symbols. In ABCD-NL, this augmentation is done via what we call a

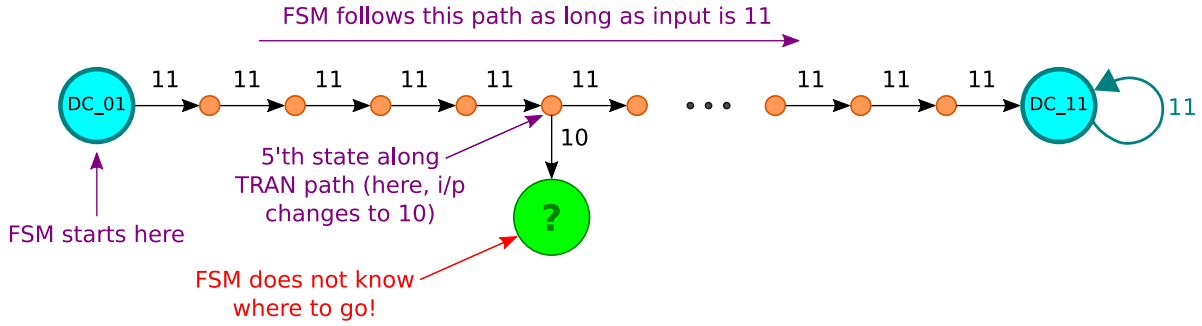


Figure 6.13: The need for a jump heuristic.

jump heuristic, *i.e.*, when a TRAN state encounters an input it was not designed for, we immediately *jump* to a state that *does* indeed know how to handle this input symbol – and the method we use to determine this state is our *jump heuristic*.

6.3.6 Two jump heuristics studied

We will now propose two jump heuristics, and we will explore their impact on the accuracy of the resulting Boolean model. The first jump heuristic (which we call `JH1`) is very simple to state and to implement; however, as we will see, there are many input sequences on which it can fail to reproduce the continuous time dynamics of the original system with good accuracy. The second jump heuristic (`JH2`) is somewhat more complicated to understand and implement, but it often produces Boolean models with much better accuracy than Boolean models that implement `JH1`.

The simplest way to understand `JH1` is using an example. Let us consider the same situation that we described before (Fig. 6.13), where the 5^{th} TRAN state along a TRAN path intended for 11 inputs did not know how to process a 10 input. `JH1` solves the problem as follows. Recall that every DC state and every TRAN state in the FSM is associated with a continuous vector of voltages and currents. Indeed, this *mapping* from the discrete FSM state to the continuous state of the given circuit is available from the SPICE simulations carried out earlier. Therefore, let us say that the DC state `DC_01` maps to the analog vector $\vec{x}_{01}$, and that the DC state `DC_11` maps to $\vec{x}_{11}$. Furthermore, let the current TRAN state (in this case, the 5^{th} TRAN state along the current path) map to $\vec{x}$. Here, `JH1` makes a simple calculation: is the current TRAN state closer to `DC_01` or `DC_11`? That is, is the vector $\vec{x}$ closer to the vector $\vec{x}_{01}$ or $\vec{x}_{11}$ respectively (for example, with closeness defined in terms of the 2-norm, as shown in Fig. 6.14)? If the former, then we can approximate the current TRAN state by the DC state `DC_01`; otherwise, we approximate the current TRAN state by the DC state `DC_11`.

Having thus found an approximation to the current TRAN state, we add an arc to the current TRAN state that has the same output symbol and the same destination FSM state as the corresponding arc emanating from the approximate state. In other words, the current

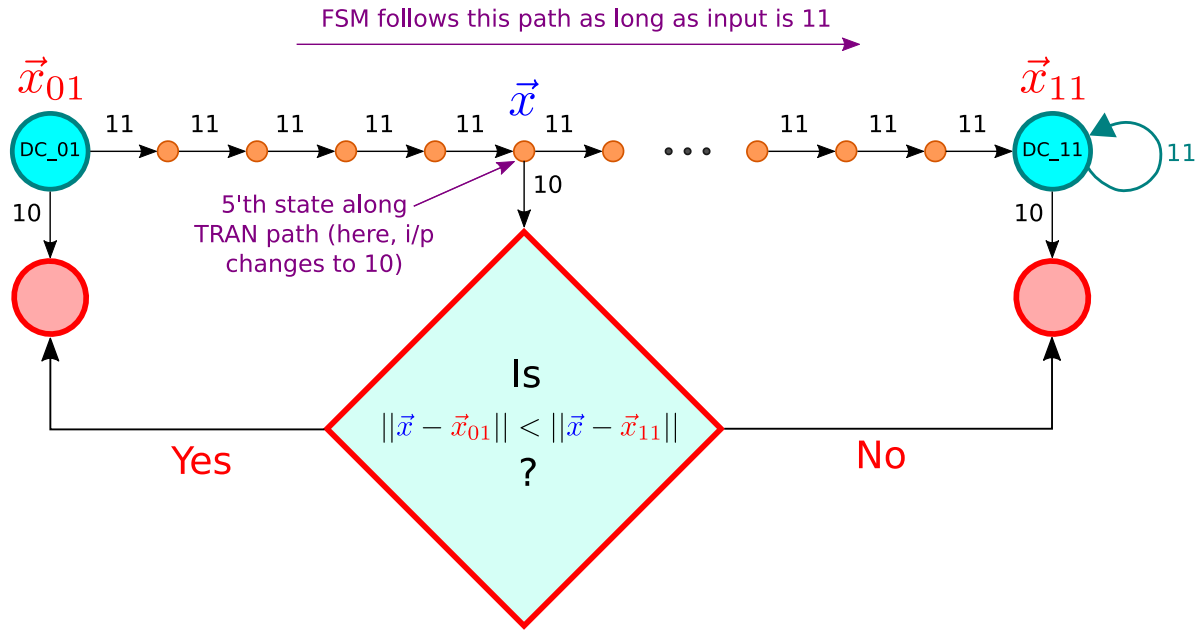


Figure 6.14: Jump heuristic 1 (or `JH1`) illustrated.

TRAN state, on any given input (other than the one it already knows how to handle), will behave exactly like the DC state that approximates it (producing the same output symbol and transitioning to the same destination state as the DC state approximation, for every input symbol other than the one it was explicitly constructed to handle). These ideas are illustrated in Fig. 6.14. Thus, the core idea behind `JH1` is: when we have TRAN state that does not know how to handle an input, we approximate the TRAN state by a DC state that we have already constructed, that *does* know how to handle the input.

By repeatedly applying the jump heuristic above to every TRAN state in the FSM, and for every "unhandled" input combination, we eventually obtain an FSM that is complete. This constitutes the Boolean model that is then returned by ABCD-NL.

Finally, we would like to point out that the calculations above involving vector norms, *etc.*, are all carried out just once, at the time the Boolean model is generated. Once the generation phase is over, one does not need to remember the mapping from the discrete domain to the continuous domain, and one does not need to make any kind of calculation involving continuous quantities – the resulting model, which will be purely Boolean in nature, would already subsume all such continuous-domain calculations.

Having presented a jump heuristic, let us now check to see how well it does in capturing the behaviour of the underlying system. To do this, we simulate both the original system and the FSM on the same input waveform/sequence, obtain the outputs predicted by both, and plot them one on top of another (for a more detailed discussion on how this is done, please see Sections 3.1 and 3.4).

Fig. 6.15 depicts a test case where the jump heuristic described above (`JH1`) works rea-

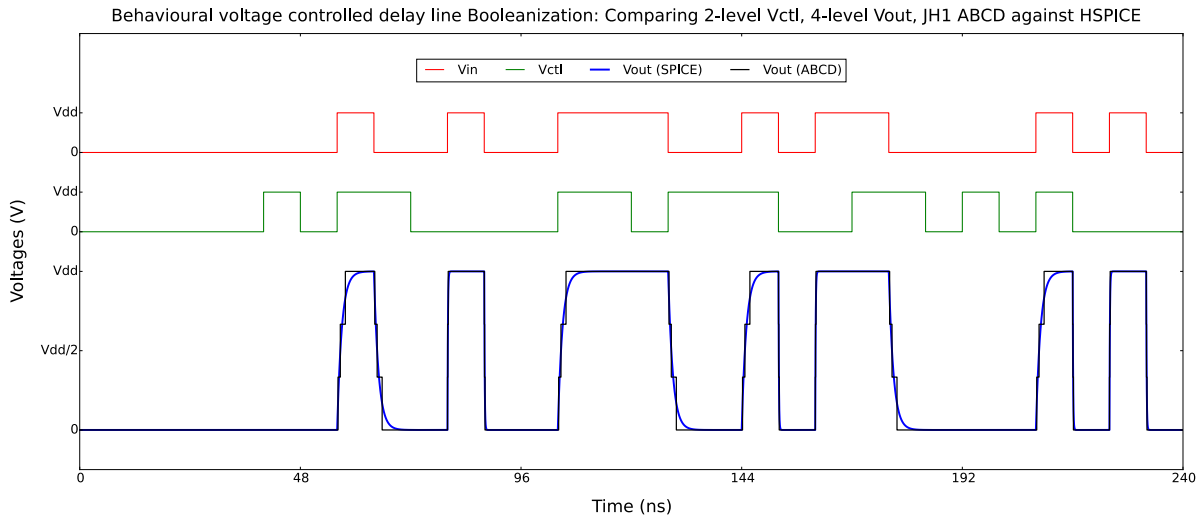


Figure 6.15: At low speed delay-line operation, `JH1` is a good enough jump heuristic.

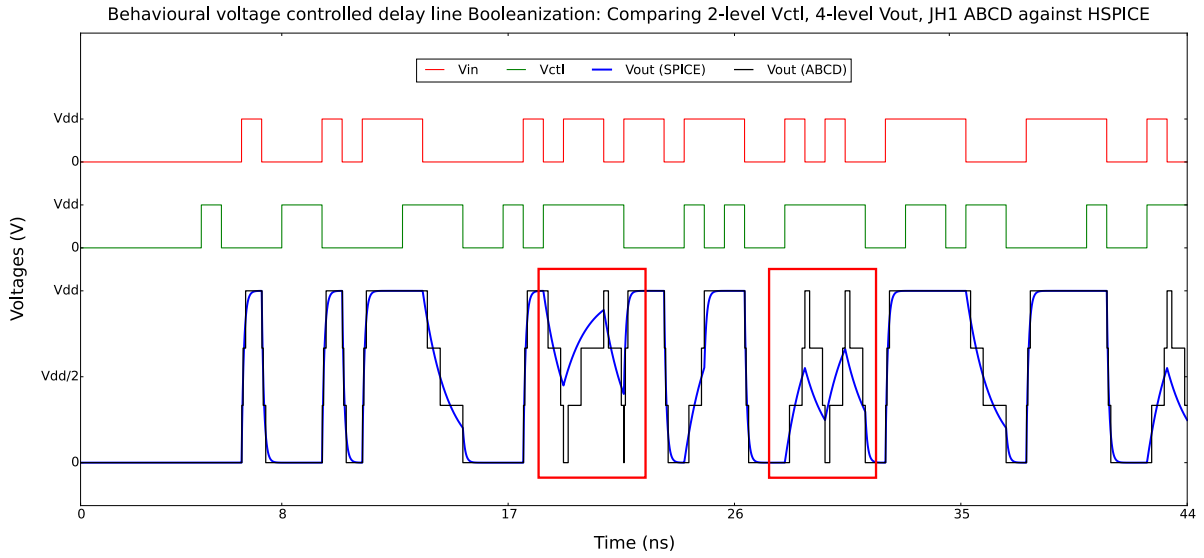


Figure 6.16: At high speed delay-line operation, `JH1` is not a good jump heuristic. The red boxes indicate where `JH1` fails to accurately capture the underlying circuit's dynamics.

sonably well. In this scenario, we have assigned a sequence of randomly generated bits to both V_{in} and V_{ctl}. However, these bits are *spaced widely apart*, or to put it in another way, the bitrate is slow. Therefore, most of the time, the jump heuristic does not even come into play because the circuit has enough time to settle to a DC state before the inputs begin to switch again. It is in such situations that a simple jump heuristic like `JH1` shines. The Boolean logic required for implementing `JH1` is very simple – so `JH1` does not significantly increase the complexity of the underlying Boolean model. At the same time, it completes the FSM as discussed above, and since it comes into play only occasionally in a low bitrate situation, it does not usually exert a significant detrimental effect on simulation accuracy.

Fig. 6.16, on the other hand, depicts a more typical test case, where `JH1` does not perform as well. Again, we have assigned a sequence of randomly generated bits to both V_{in} and V_{ctl}. But this time, bits are *spaced closely together*, or to put it in another way, the bitrate is fast. Therefore, a lot of the time, the circuit does not have enough time to settle to a DC state before the inputs change: the circuit is always in a state of flux, and the FSM model is constantly jumping from one TRAN path to another. Therefore, the accuracy of the jump heuristic makes a big difference in this case, and a simplistic approach like `JH1` is found to be inadequate. For example, the predictions made by `JH1` for the circuit do not always tally well with the continuous dynamics of the circuit: the red boxes in Fig. 6.16 call attention to such discrepancies.

Thus, the jump heuristic `JH1` is not well-suited for high input bitrates. We need a new jump heuristic that is capable of handling such inputs more gracefully and accurately. This is our motivation for developing `JH2`, our next jump heuristic.

JH2 works as follows: instead of approximating the current TRAN state by a DC state (as `JH1` does), `JH2` approximates the current TRAN state using either another TRAN state or a DC state. This is illustrated in Fig. 6.17. Let us consider the same problem as before: let us say that the FSM is in the 5^{th} TRAN state along the TRAN path leading from DC state `DC_01` to DC state `DC_11`. At this state, let us say that the FSM encounters the input symbol 10, which the FSM does not know how to respond to. With `JH1`, the FSM would simply mimic the behaviour of either `DC_01` or `DC_11`, whichever happens to be "closer" to the current TRAN state. In `JH2`, we extend this idea to a broader set of possible "matching states" that already know how to handle the input symbol 10. As Fig. 6.17 shows, these "candidate matching states" include all DC and TRAN states along the TRAN path from `DC_01` to `DC_10`, as well as all the DC and TRAN states along the TRAN path from `DC_11` to `DC_10`. Note that all these candidate matching states already know how to handle the input symbol 10. In `JH2` therefore, we first determine the candidate matching state that "best" matches the current TRAN state: this is the candidate state whose analog vector of voltages and currents is closest to that of the current TRAN state, where closeness is measured by the 2-norm. Having determined such a "best matching" state, we then add an arc to the current TRAN state for the input symbol 10; this arc simply mimics the behaviour (in terms of destination FSM state and output symbol) of the best matching state. As with `JH1`, we repeat this process for all TRAN states until the FSM is complete.

Fig. 6.18 illustrates the idea above on a couple of states belonging to the TRAN paths

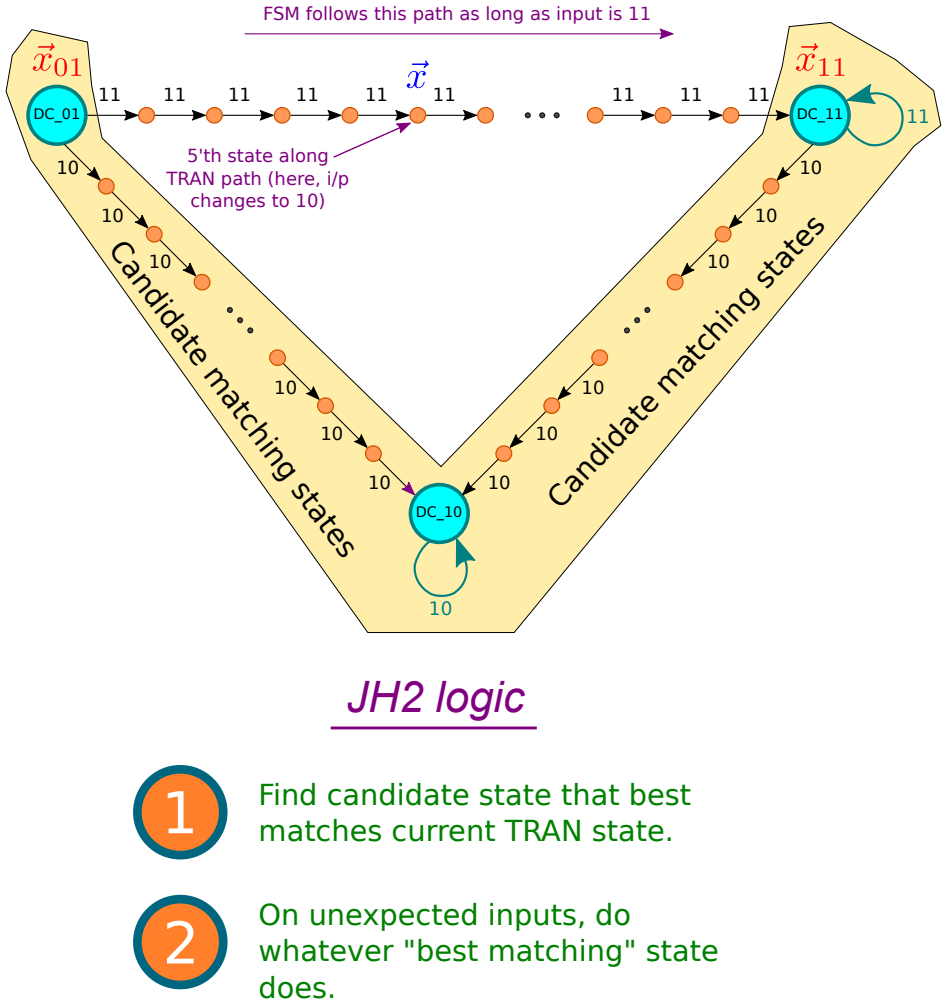


Figure 6.17: Jump heuristic 2 (or JH2) illustrated.

depicted in Fig. 6.12. Let us consider what the FSM does when it encounters the unexpected input 00 while it is in the orange state along the upper TRAN path shown in the figure. Clearly, this input is encountered by the FSM in the midst of a transition from DC_00 to DC_10 on input 10. The analog voltage V_{out} corresponding to this orange state is about 0.5866V (this is obtained from one of the SPICE simulations carried out earlier). The key idea behind JH2 is to identify that state along the DC_10 $\rightarrow$ DC_00 path whose analog value is closest to 0.5866V. By looking through our previously run SPICE simulations again, we determine that this happens to be the 5^{th} state along this path, which has an analog value of 0.5770V. Thus, when the orange state encounters the unexpected input 00, it does whatever this 5^{th} state does on input 00. This action happens to be to jump to the 6^{th} state along the DC_10 $\rightarrow$ DC_00 path while emitting the output symbol 2. Therefore, we add an FSM arc (shown in blue in Fig. 6.18) originating in the orange state, with input symbol 00, output

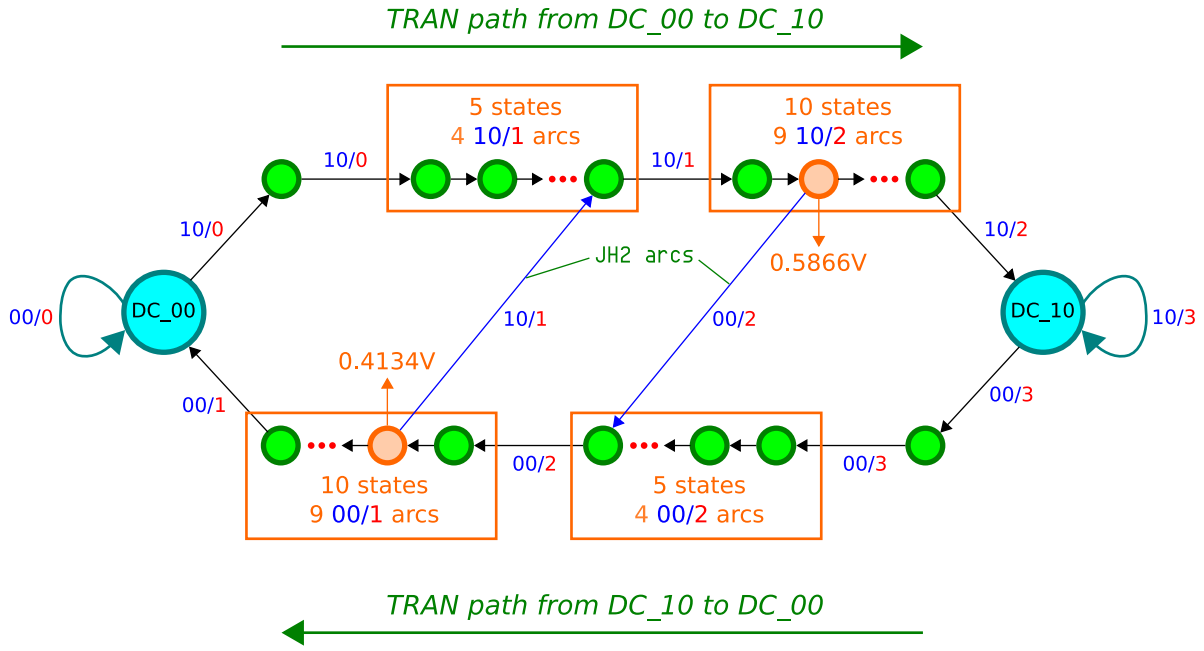


Figure 6.18: Jump heuristic `JH2` illustrated on a couple of FSM states belonging to the TRAN paths of Fig. 6.12.

symbol 2, and destination the 6^{th} state along the `DC_10` $\rightarrow$ `DC_00` path. In this way, we complete the FSM by repeating this process for every unexpected input encountered at every TRAN state in the FSM. Fig. 6.18 also shows a `JH2` arc added to an orange state on the lower `DC_10` $\rightarrow$ `DC_00` TRAN path.

We note that the jump heuristic `JH2` is not as simplistic as `JH1`. It involves considering many more candidate matching states than `JH1`, which only involves two possible matches (the DC states at either end). Consequently, a Boolean model that implements `JH2` takes longer to generate, and the resulting Boolean logic is also more complicated (and can involve many more logic gates or other constructs to implement). However, in return for this additional generation time and complexity, we usually get excellent accuracy. For example, Fig. 6.19 shows that `JH2` is indeed able to accurately capture the behaviour of the delay line above for both slow and fast input bitrates (in contrast to `JH1`, which only worked well for low bitrates).

Indeed, this result holds true for higher levels of discretization as well. For example, Fig. 6.20 shows a scenario where the delay line's output V_{out} is discretized using 8 quantized levels instead of 4. In this case also, at high bitrates, we see that `JH2` is able to capture the behaviour of the circuit much more accurately than `JH1`. For example, the red boxes in the top half of the figure indicate discrepancies between the original (continuous) system's output and the output predicted by an FSM using `JH1`. The bottom half of the figure (which uses `JH2`), on the other hand, does not contain any such discrepancies.

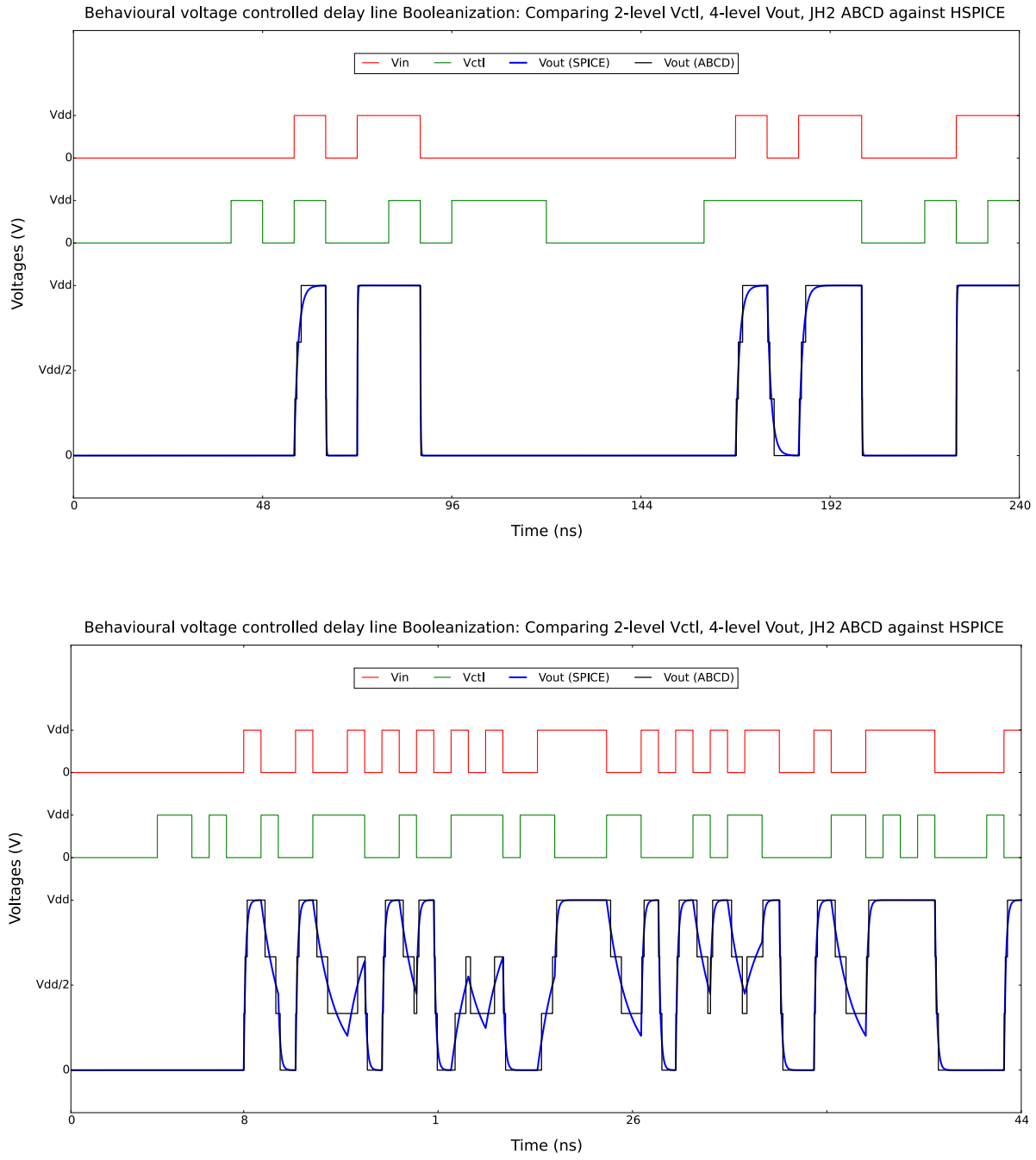


Figure 6.19: At both low speed and high speed delay-line operation, JH2 is a good jump heuristic: it captures the continuous dynamics of the given circuit with high accuracy.

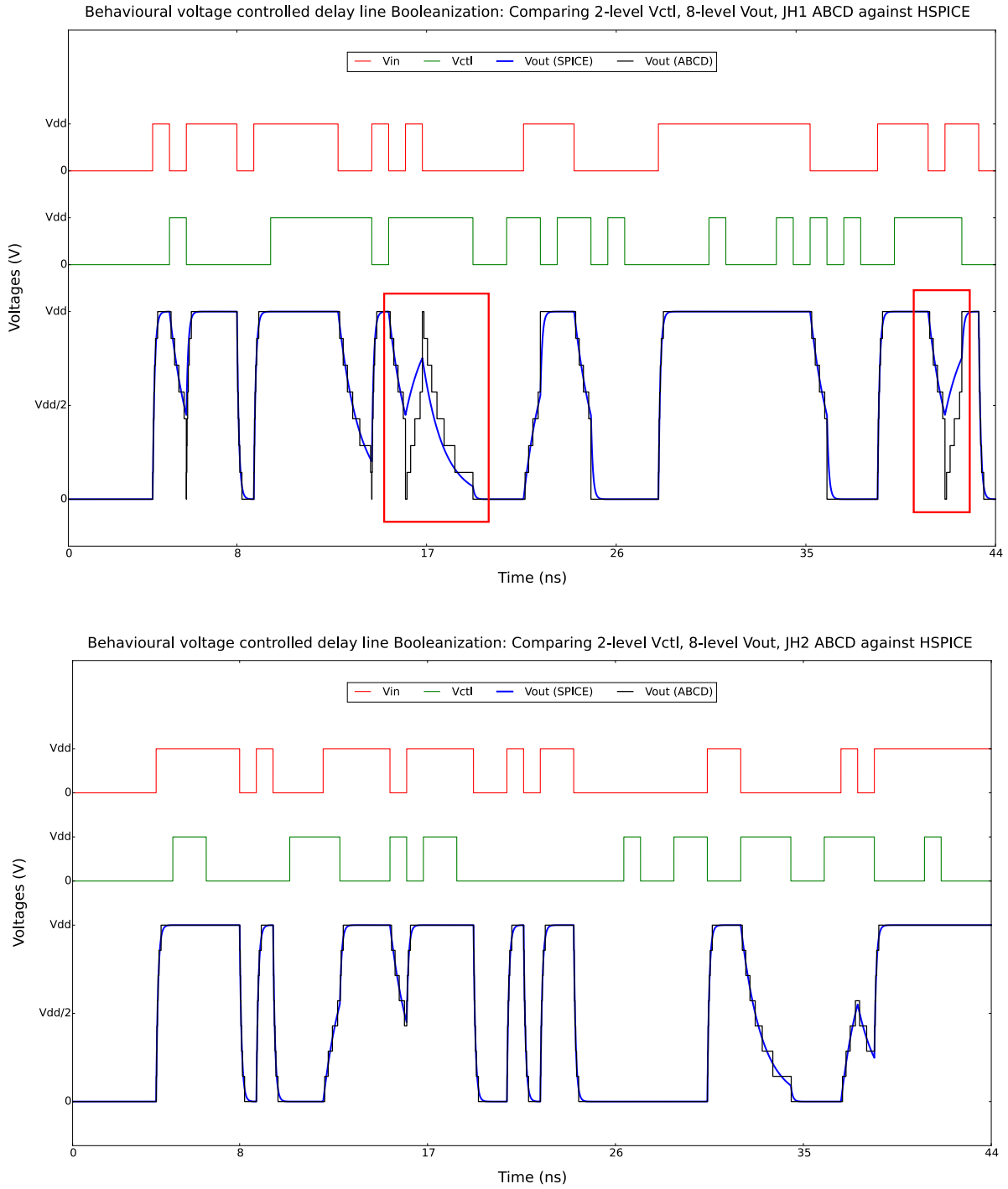


Figure 6.20: `JH2` continues to be a better jump heuristic than `JH1` at high input bitrates, even when the number of levels used to discretize V_{out} is increased from 4 to 8.

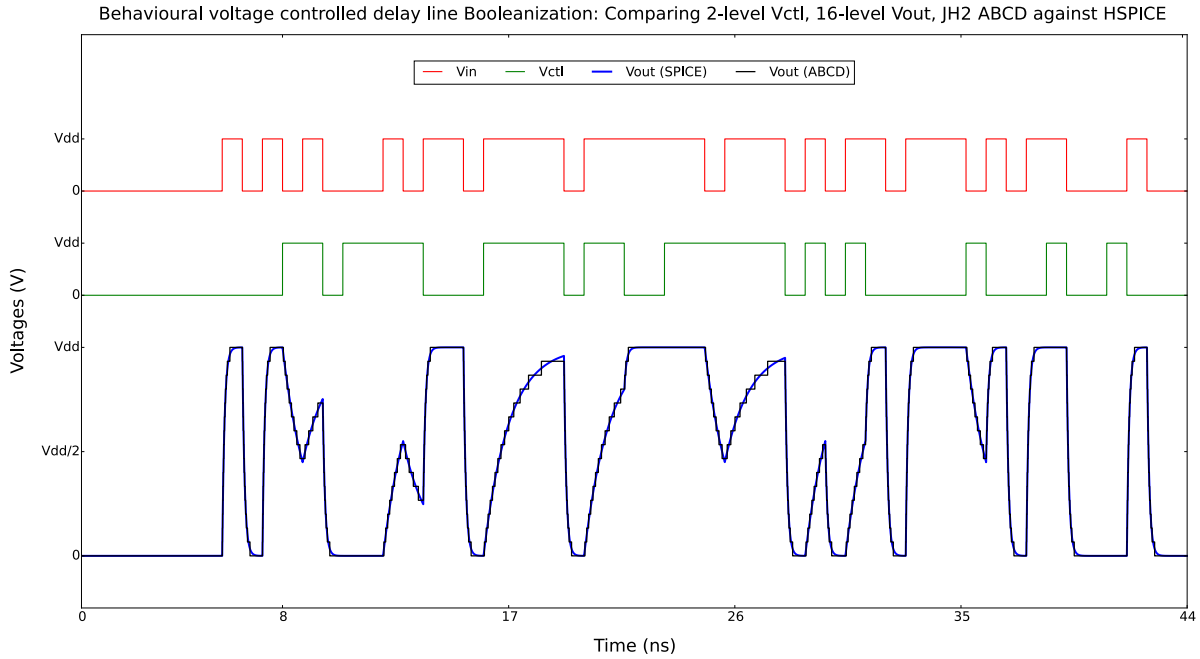


Figure 6.21: JH2 continues to be a good jump heuristic when the number of levels used to discretize V_{out} is increased from 8 to 16.

Fig. 6.21 increases the number of levels used to discretize V_{out} even further – from 8 to 16. As seen from the figure, at this level of discretization, JH2 performs very well indeed: the waveforms produced by an FSM implementing JH2 are able to closely track the continuous-time waveforms predicted by SPICE, even for high input bitrates.

Finally, we increase the number of levels used to discretize the control voltage V_{ctl}. Up until this point, we have used just 1 bit to represent the control voltage V_{ctl}; consequently, V_{ctl} could only assume 2 distinct values. Now we allow V_{ctl} to take on 4 different values – by discretizing it using 2 bits instead of 1. This, of course, changes several aspects of the ABCD-NL FSM. For example, the FSM now has 8 DC states instead of 4. The number of SPICE simulations required to build the FSM increases to $8^2 - 8 = 56$ from just 12, and so on.

Fig. 6.22 shows the results. The figure contains 3 sub-plots; the top sub-plot discretizes the output voltage V_{out} using 4 levels, the middle one uses 8 levels, and the bottom one uses 16 levels. In each case, the control voltage V_{ctl} is discretized using 4 levels. In all three cases, as shown in Fig. 6.22, the Boolean models generated by ABCD-NL (using JH2 as the jump heuristic) are able to capture the SPICE-level dynamics of the given circuit to high accuracy. Indeed, as the number of levels used to discretize V_{out} increases, it becomes difficult to visually distinguish between the continuous output waveform predicted by SPICE and the discrete output waveform predicted by ABCD-NL's FSMs.

We now briefly touch upon a third jump heuristic (JH3), a small tweak of JH2 which, for

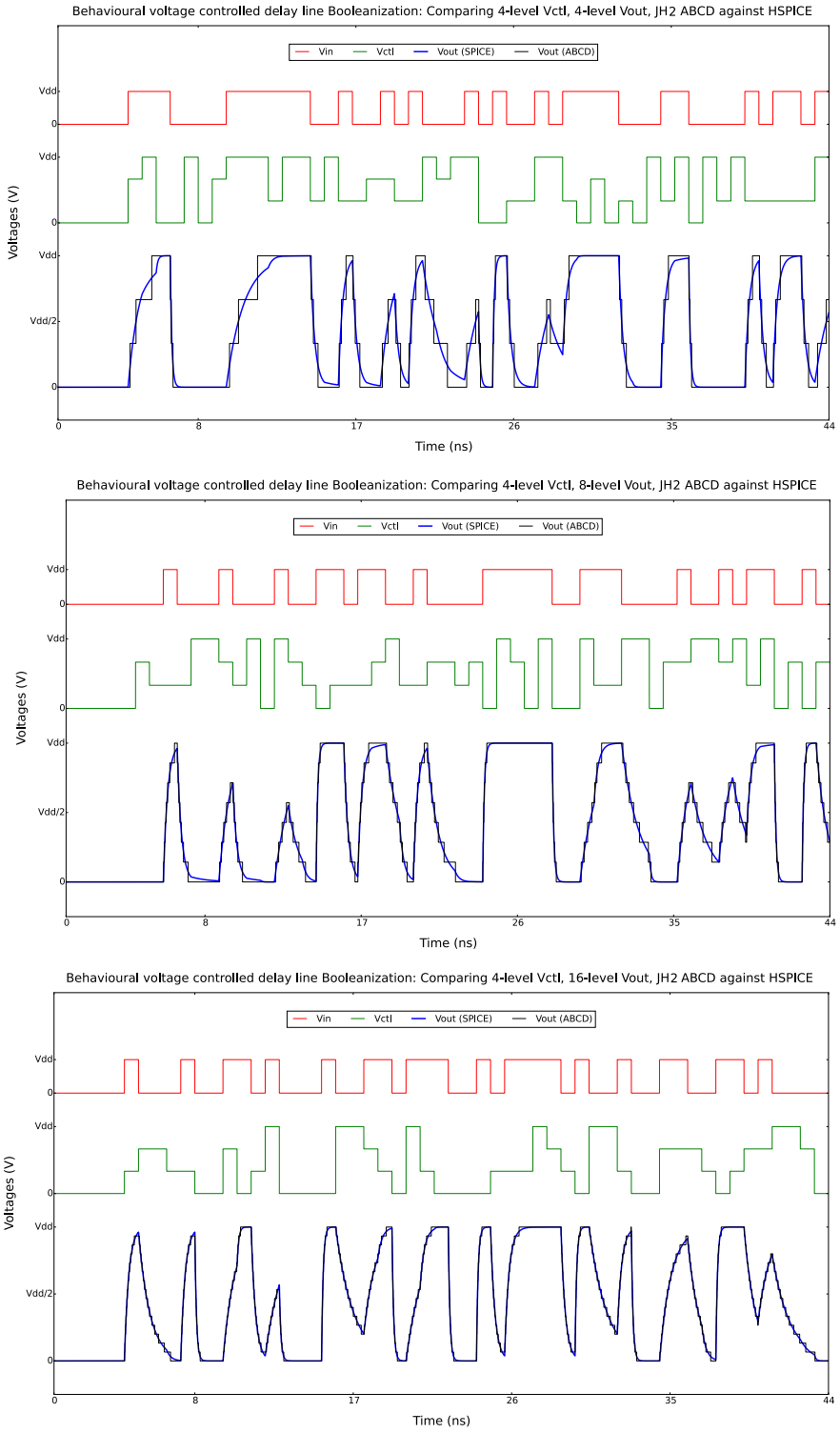


Figure 6.22: Comparing the predictions made by an FSM using `JH2` against those made by SPICE, when V_{ctl} is discretized using 4 levels, and V_{out} is discretized using 4, 8, and 16 levels respectively. In each case, the FSM's predictions are able to match SPICE with good accuracy, which demonstrates the effectiveness of `JH2` as a jump heuristic.

some circuits (such as D-Latches and analog comparators), can not only improve the accuracy of the generated Boolean model, but also significantly reduce the time taken to generate it. Recall from the discussion above that `JH2`, while finding the "best matching state" v on TRAN path p, for a given TRAN state u, compares the analog state vector of voltages and currents $\vec{x}_u$ associated with u against *every* state on path p. `JH3` is different in that it does not do this comparison for every state on path p; instead, it only compares $\vec{x}_u$ with those FSM states on path p that have the same discretized output (obtained directly from the respective analog state vectors) as u. This ensures that the Boolean model maintains some *output continuity* while jumping from one TRAN path to another. It also prevents voltages/currents that do not directly determine the output from exerting undue influence on the norm calculations carried out by the heuristic. Finally, by restricting the number of candidate states that can be "best matches", `JH3` eliminates a number of norm calculations that would be carried out by `JH2`, and hence often runs much faster than `JH2`. However, for some circuits (*e.g.*, charge pumps), we found that `JH2` is the more accurate jump heuristic. We still do not clearly understand exactly when `JH2` would be more accurate than `JH3`, or vice-versa; while Booleanizing an unknown AMS system, we usually try both heuristics and pick the one that matches SPICE simulations better.

Having described the core techniques used by ABCD-NL to Booleanize genuinely non-linear AMS designs, and having provided a detailed behavioural example illustrating every step of the ABCD-NL algorithm (including discussions about why naïve Booleanization procedures such as Backward Euler discretization can fail due to artifacts such as fake fixed points and unphysical discontinuous discrete jumps), we now apply ABCD-NL to Booleanize SPICE-level real-world circuits that are of interest to AMS designers. These include: (1) a charge pump/filter system that is relevant to PLL design, (2) a signalling/communication sub-system that involves (non-linear) digital logic interfacing with an analog channel, (3) a delay line that also plays an important rôle in circuits such as PLLs and DLLs, (4) a D/A converter, and, (5) an analog comparator, the last two circuits being important analog components that make up the SAR-ADC shown in Fig. 6.2. In each case, we show that the Boolean model produced by ABCD-NL is able to accurately reproduce the SPICE-level analog dynamics of the underlying circuit, including important performance-limiting non-ideal analog phenomena.

6.4 Example: Booleanizing a charge pump driving an analog filter

Fig. 6.23 shows a charge pump driving an analog filter, a system that plays an important rôle in PLLs. The system works as follows: the transistors M1 and M2 form a current mirror that can pump a current I_0 into the load capacitor C_L, whereas transistors M3 and M4 form an opposing current mirror that can withdraw current I_0 from C_L. The circuit has two inputs, V_{up} and V_{down}. During normal operation, exactly one of these inputs is high; if V_{up} (V_{down})

is high, current is pumped in (out), driving the output voltage V_{out} higher (lower); this is called the charging (discharging) mode of the charge pump, and the system responds most quickly when in one of these modes (*e.g.*, using a 90nm process, response times are typically of the order of tens of nanoseconds).

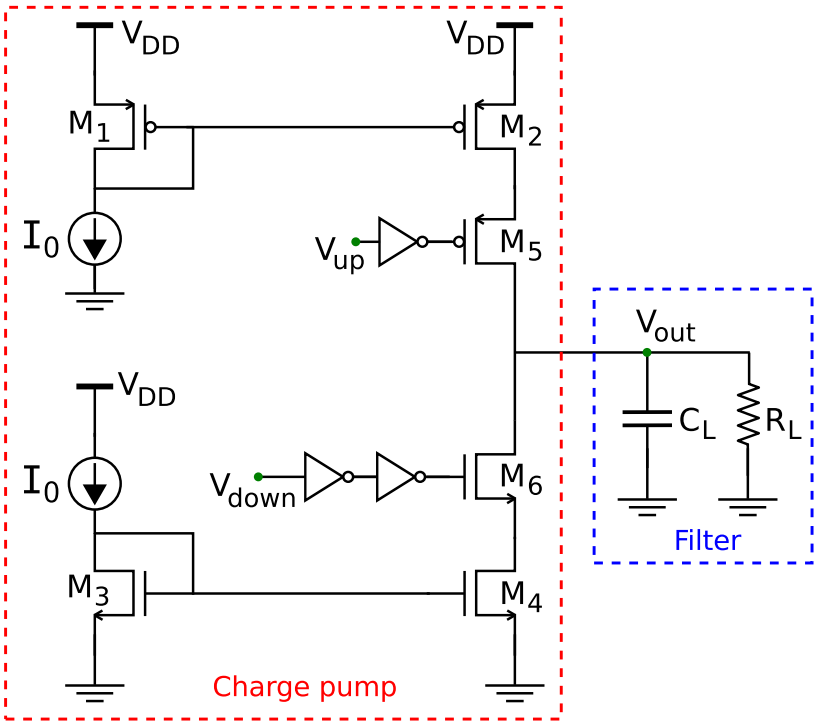


Figure 6.23: Schematic of a charge pump driving an analog filter.

In addition to the normal mode of operation, it is also important to capture the behaviour of the charge pump under anomalous inputs; for example, if both inputs are high, the charge pump enters an imbalance-driven mode, which can either charge or discharge the load, depending on operating conditions. This type of charging/discharging is typically much slower than normal operation (*e.g.*, hundreds of nanoseconds response time), because only a small current flows through the load. Finally, if both inputs are off (cutoff mode), the output voltage, under ideal conditions, would remain constant; however, due to leakage currents and the resistive load R_L, the capacitor C_L slowly discharges to a DC voltage that is almost 0. The cutoff mode is the slowest mode of operation of the system, with response time in the microsecond range.

Since PLL performance critically depends on non-linear charge pump/loop-filter dynamics, our goal is to use ABCD-NL to accurately model the behaviour of the above system under all possible operating conditions: charging, discharging, imbalance-driven, and cut-off. So we designed the system in 90nm CMOS, using BSIM4 device models. We then applied Algorithm 1 (of Section 6.2) to Booleanize this system (using 5 bits to encode the

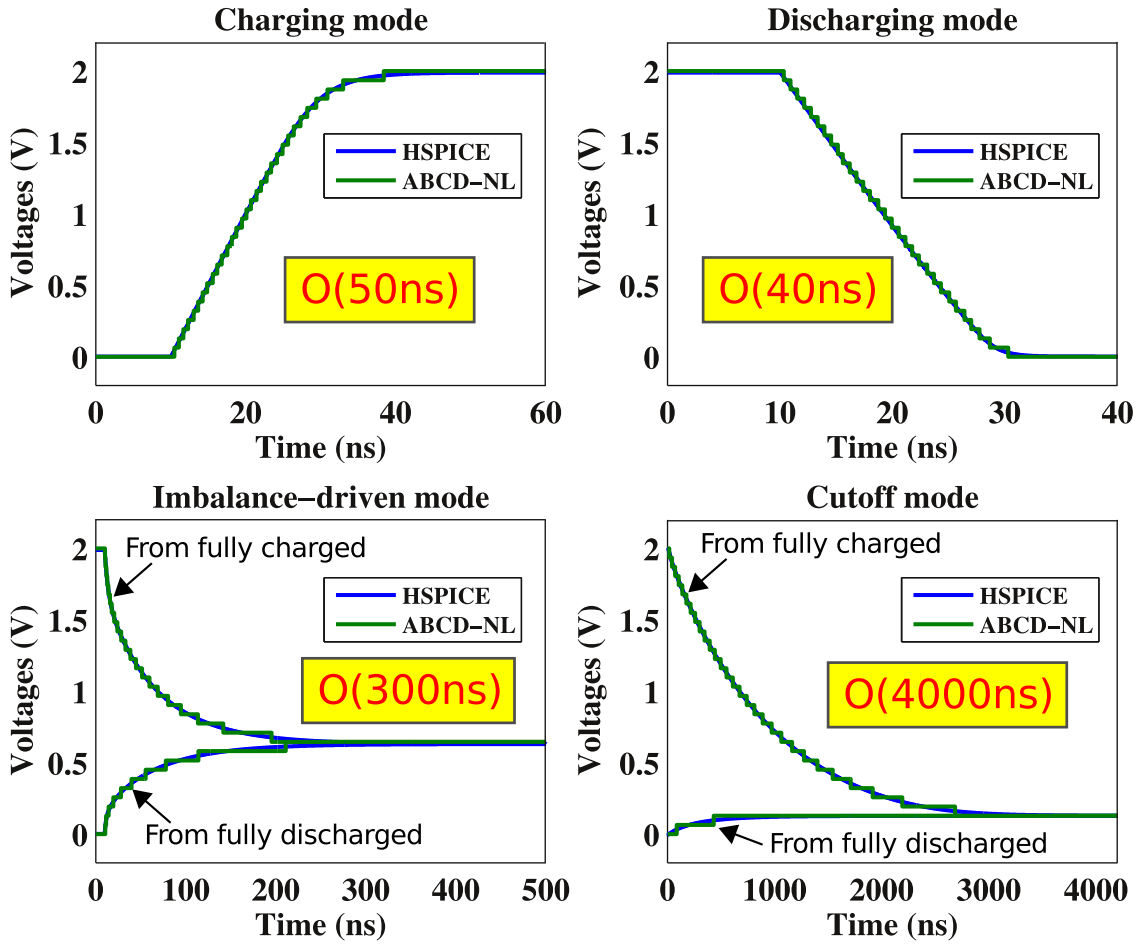


Figure 6.24: ABCD-NL accurately captures the behaviour of the charge pump under all four modes of operation, in spite of the widely differing time scales involved.

output waveform), and we simulated the resulting Boolean model using Algorithm 2, on a range of inputs that covered all four modes of operation. In each case, we compared the output predicted by ABCD-NL's Boolean model, against that predicted by HSPICE. Fig. 6.24 shows the results, where HSPICE waveforms are shown in blue, and ABCD-NL waveforms are in green. As the figure shows, in spite of the widely differing time scales involved in the four modes, the Boolean model produced by ABCD-NL closely matches the SPICE-level dynamics of the system in all its modes.

Fig. 6.25 further demonstrates the accuracy and robustness of the Boolean model produced by ABCD-NL. The figure shows a long pseudo-random bit sequence applied as input to the circuit, which switches the circuit in and out of all 4 modes of operation over a long time frame. Throughout this time, it is seen that the Boolean model produced by ABCD-NL (the green waveform) closely tracks the SPICE-simulated output (the blue waveform) of the system. This indicates that ABCD-NL is indeed a powerful and accurate modelling tech-

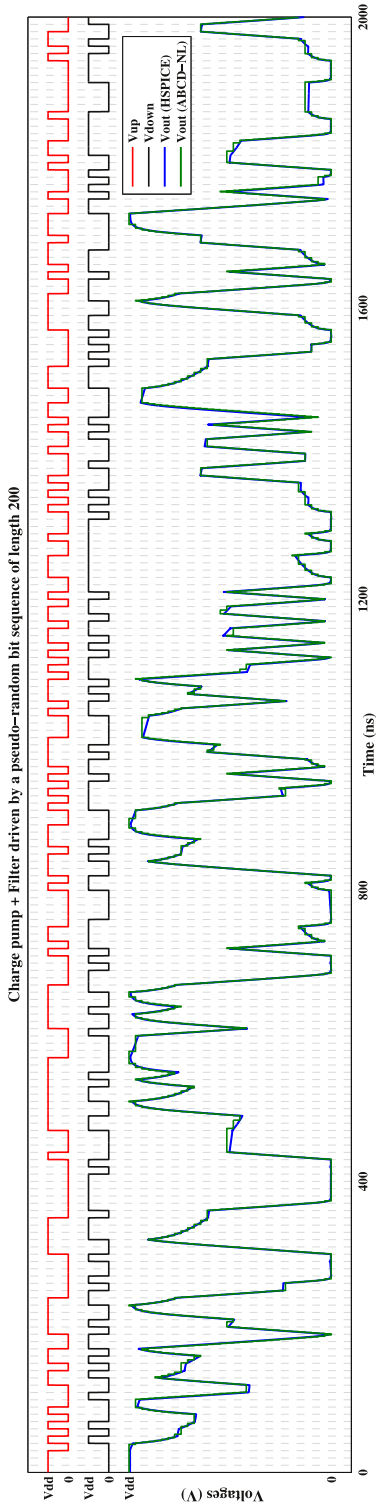


Figure 6.25: ABCD-NL can predict the response of the charge pump/filter system accurately (at least from a visual inspection or "eyeball metric" perspective), even over long time frames that involve rapid switching between all 4 modes of operation.

nique, and one that can conceivably be used as a much faster, almost-as-accurate, drop-in replacement for SPICE over long transient runs.

6.5 Example: Booleanizing a signalling system (non-linear digital logic + LTI analog channel)

Fig. 6.26 shows a mixed-signal sub-system, of the type depicted in Fig. 6.1. As we mentioned in Section 6.1, such systems are often encountered in Signal Integrity (SI) applications. In this example, the transmit side takes 3 bits as input (A, B, and C_{in}), and adds them up using a full-adder, thereby producing 2 output bits: the sum S, and the output carry C_{out}. These 2 bits are then sent across an analog channel that consists of several RC stages, inter-linked with coupling capacitances. At the other end of the channel, the receiver cleans the arriving waveforms S_{ch} and C_{ch} using chains of inverters.

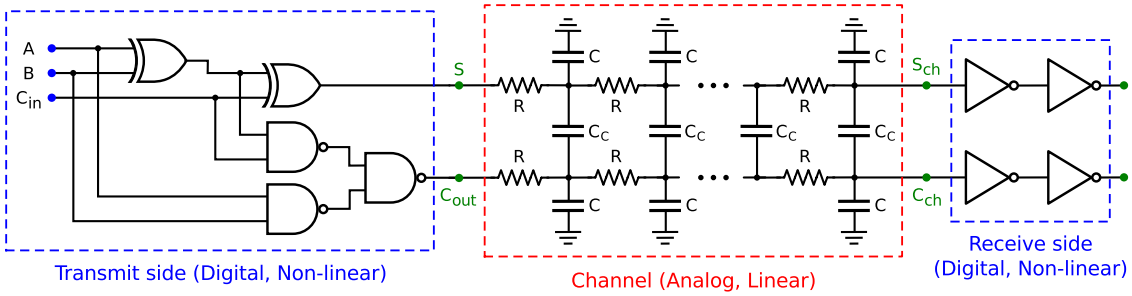


Figure 6.26: A signalling/communication sub-system that arises in SI applications.

We have modelled each transistor in the above system using a 22nm BSIM4 analog SPICE model (obtained from [51]). Therefore, the system above exhibits several realistic non-linear analog effects, including leakage currents, loading effects, delays, channel-induced ISI/crosstalk, *etc.*. Our goal is to use ABCD-NL to accurately reproduce the analog waveforms at the output of the channel (S_{ch} and C_{ch}), in the presence of these adverse analog effects. This is a crucial requirement for SI analysis.

Fig. 6.27 shows the results obtained by applying ABCD-NL to the above system. Part (a) shows three randomly generated 40-bit sequences (for A, B, and C_{in}, respectively), applied as inputs to the circuit. In part (b), these inputs are applied at a bitrate of 1 Gbps. At this bitrate, the system behaves in a fairly ideal manner, *i.e.*, distortion, crosstalk, *etc.*, are minimal, as seen from the blue HSPICE waveforms S_{ch} and C_{ch} of Fig. 6.27 (b). Also, the green waveforms in the figure show the predictions made by ABCD-NL's purely Boolean model (using 4 bits to encode each output waveform); as the figure shows, ABCD-NL's purely Boolean model is able to accurately predict the system's dynamics at this bitrate.

Fig. 6.27 (c) shows the same bit pattern applied to the circuit, but at a higher bitrate (3.2 Gbps). At this bitrate, the system produces considerable dispersion (due to ISI and crosstalk

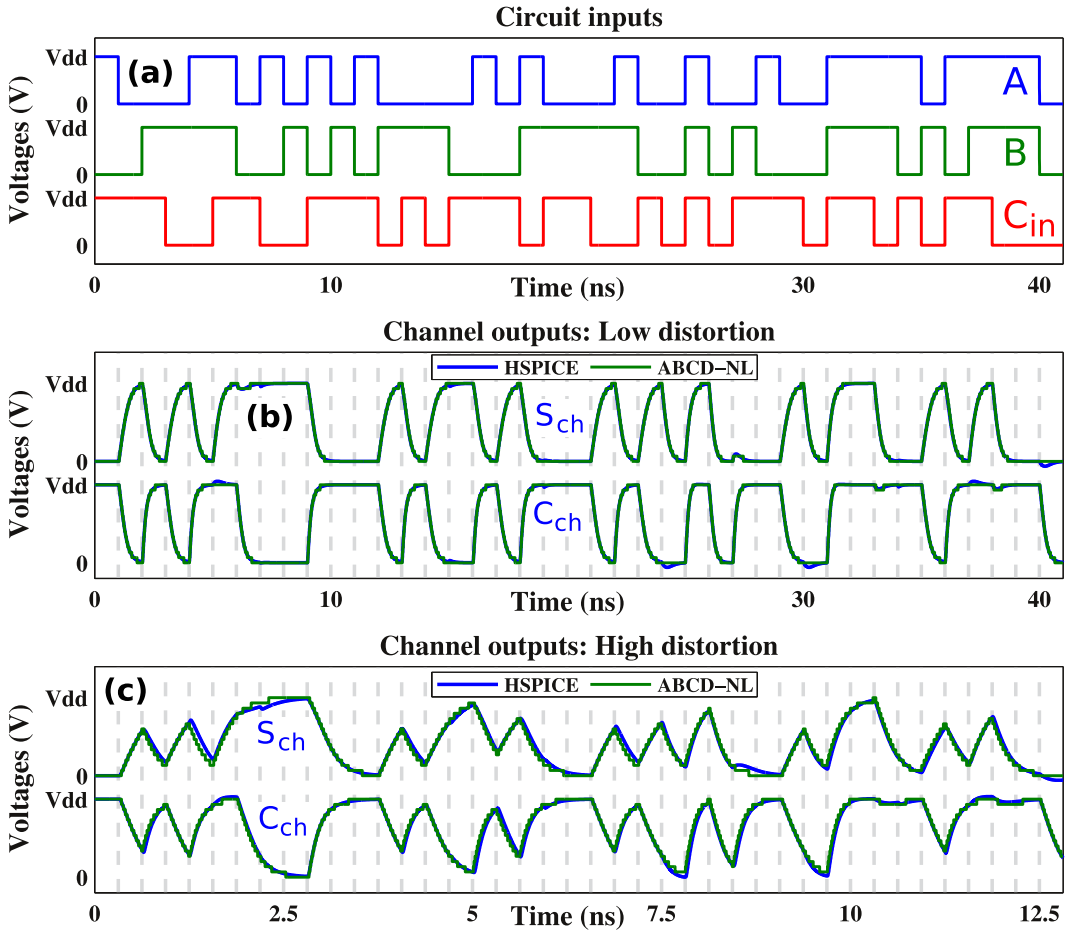


Figure 6.27: ABCD-NL closely matches the SPICE-level analog behaviour of the signalling/communication sub-system of Fig. 6.26, at both high and low bitrates.

from the channel), in addition to the distortion produced by the non-linear dynamics of the transmit and receive sides. Here also, it is seen from the figure that ABCD-NL is able to accurately reproduce the analog dynamics exhibited by the system.

6.6 Example: Booleanizing a voltage controlled delay line

In this section, we use ABCD-NL to Booleanize a voltage controlled delay line – a circuit that finds application, for example, in delay locked loops (DLLs) and phase locked loops (PLLs). As Fig. 6.28 shows, the delay line has two inputs: the data bits V_{in}, and the control voltage V_{ctl}. The output (V_{out}) is a delayed version of the input bits, and the delay between the input and the output is controlled using V_{ctl}. In the current-starved topology shown

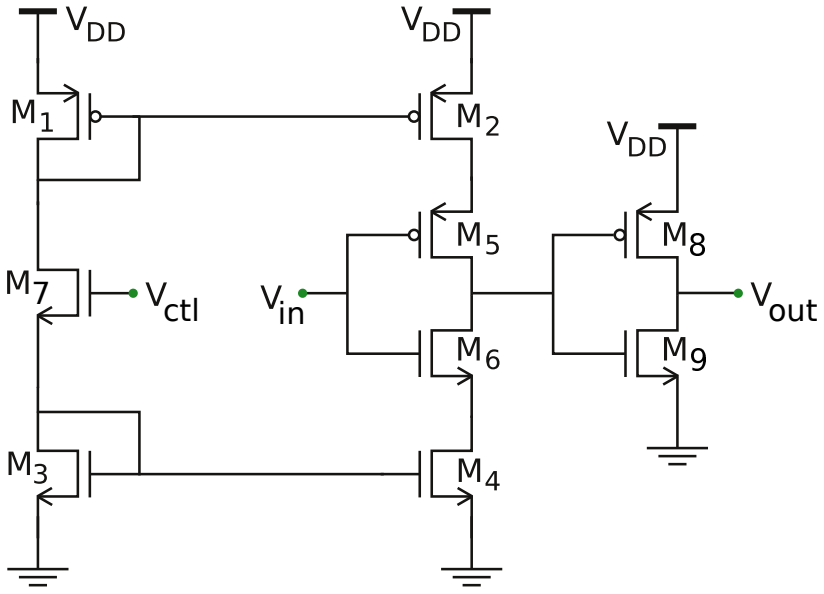


Figure 6.28: Schematic of a voltage controlled delay line.

in Fig. 6.28, higher values of V_{ctl} result in smaller delays, and vice-versa.

We designed the above circuit in 90nm CMOS (with BSIM4 device models) and Booleanized it using ABCD-NL, discretizing the output using 4 bits. To validate the Boolean model, we applied a randomly generated sequence of bits at the input of the delay line, while varying the control voltage sinusoidally. As Fig. 6.29 shows, the propagation delay between the input and the output is high when V_{ctl} is low, *i.e.*, near the sinusoidal minima. At such times, for instance, the output voltage V_{out} does not have enough time to rise/fall all the way to $V_{DD}/0$ before the next data bit arrives. At other times (*e.g.*, near the sinusoidal maxima), however, the propagation delay is small. And as the figure shows, ABCD-NL's Boolean model is able to accurately capture the circuit's dynamic delay variations over extended periods of time.

6.7 Example: Booleanizing a D/A converter (DAC)

We now apply ABCD-NL to produce a purely Boolean model of a canonical mixed-signal system, a D/A converter used within a SAR-ADC. As Fig. 6.30 shows, we have a 4-bit D/A converter consisting of four Analog Devices AD8079A analog buffers (SPICE models available from [71]), and an R/2R ladder that feeds into a voltage follower.

A key figure of merit of a D/A converter is speed; so it is important to accurately capture the delay of the system for all possible bit transitions at the input. Fig. 6.31 shows many of these transitions (to avoid unnecessary repetition, we do not show all the transitions), and indeed, it can be seen from the figure that ABCD-NL accurately captures the system's delay for all these inputs (using 6 bits to encode the D/A output).

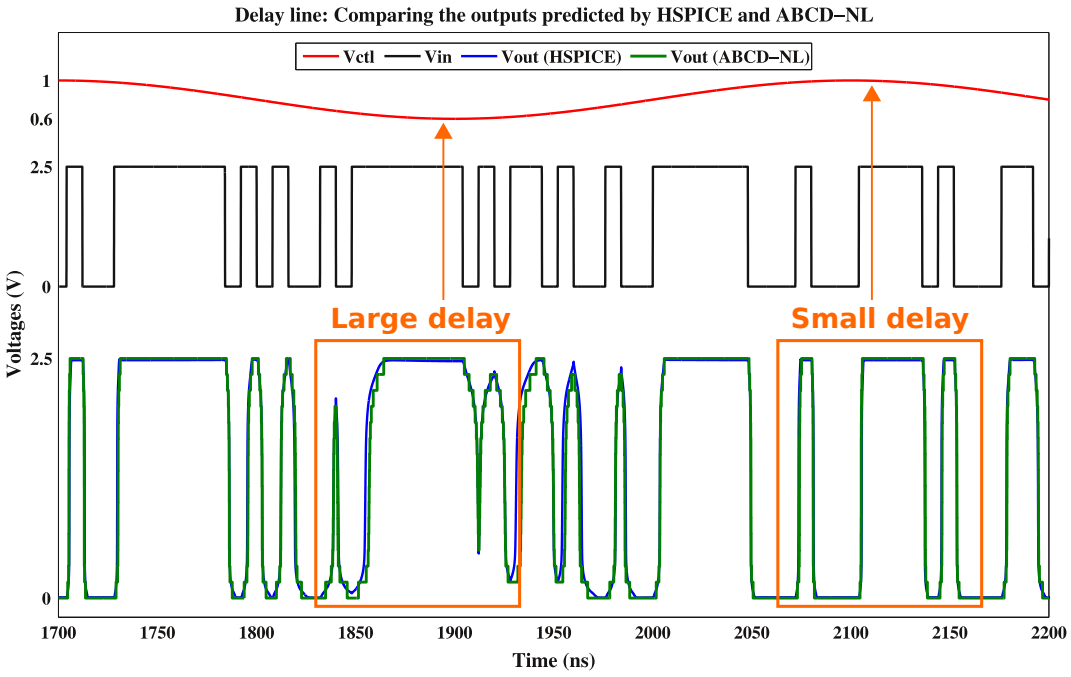


Figure 6.29: ABCD-NL accurately reproduces the SPICE-level behaviour of a delay line.

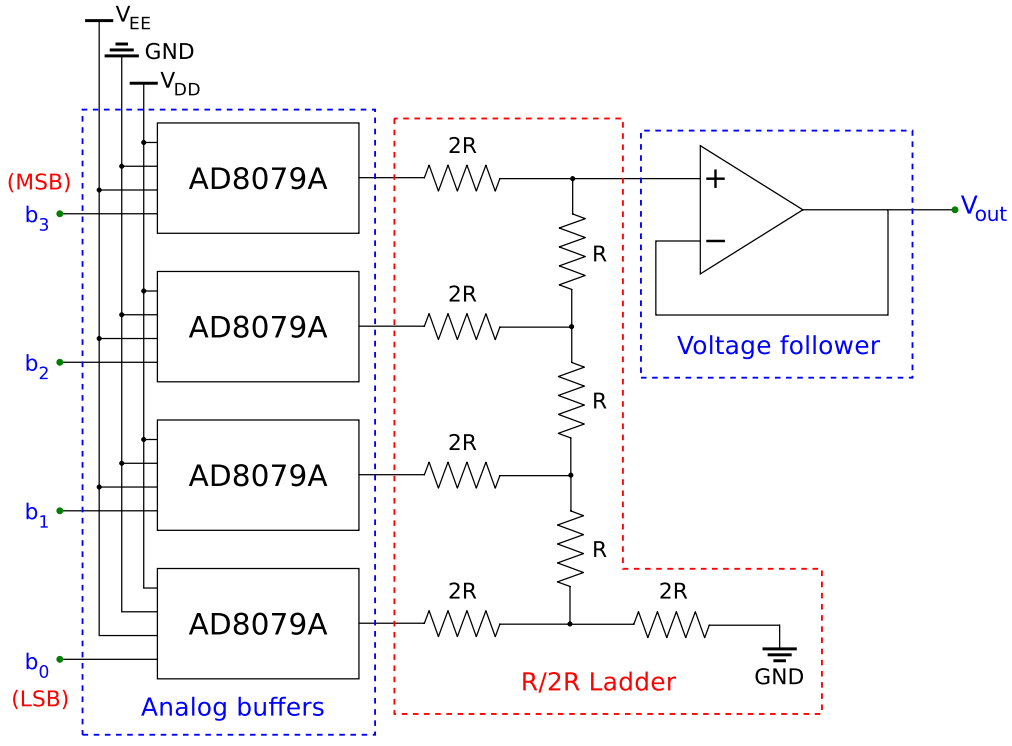


Figure 6.30: Schematic of a D/A converter used within a SAR-ADC.

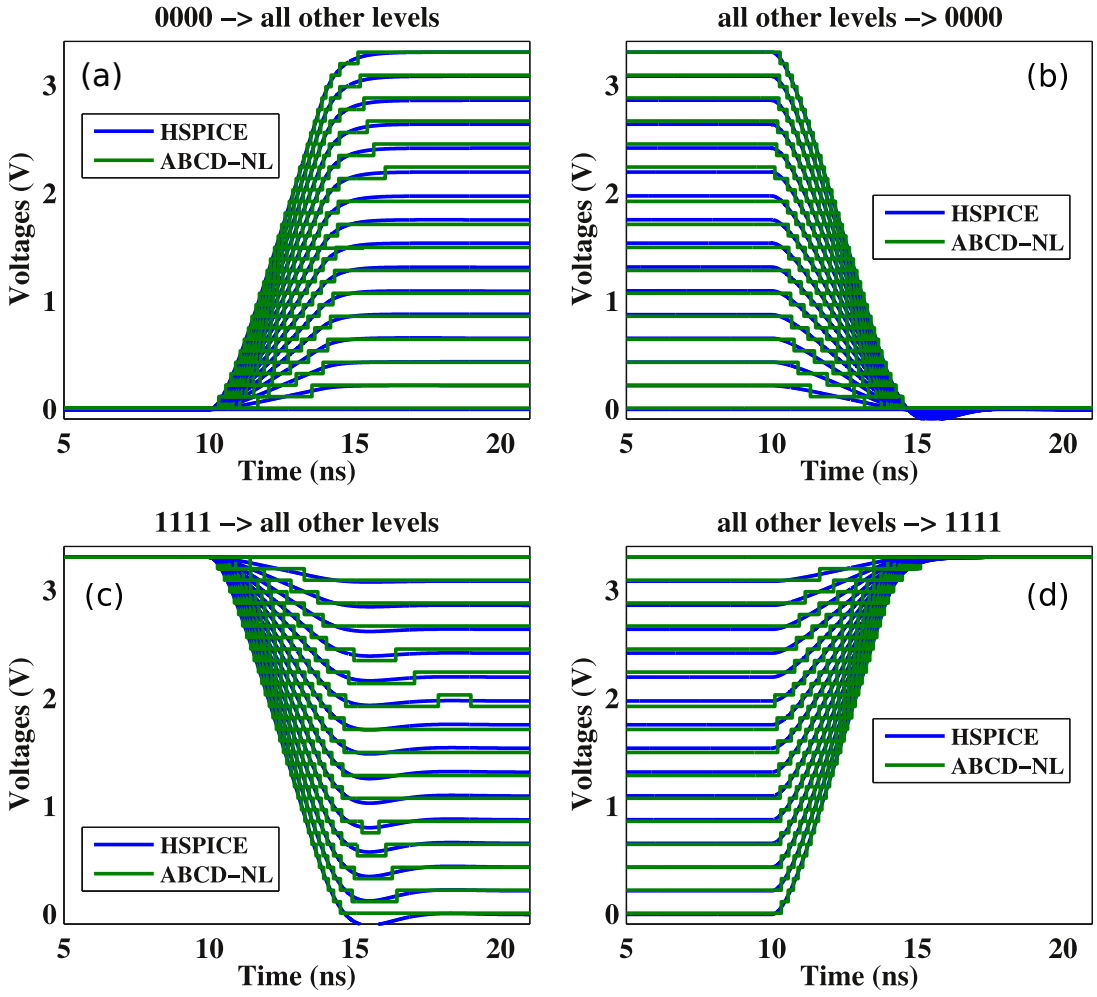


Figure 6.31: ABCD-NL closely matches SPICE-level simulation of the D/A converter, for several input bit transitions.

Furthermore, because our D/A converter is embedded within a SAR-ADC, it is important to have our Boolean model reproduce the system's dynamics for input patterns that are typical to the SAR-ADC environment. Fig. 6.32 illustrates this environment for a 4-bit SAR-ADC. The red waveform shows a 150kHz sine wave, which is the ADC input. The ADC operates at about 8MHz, so each period of the input generates about 52 ADC samples. Over these samples, the input bits b_0 to b_3 of the D/A converter switch as shown in the top half of Fig. 6.32. The blue waveform at the bottom of the figure depicts the D/A output, as predicted by HSPICE. And as seen from the green waveform, ABCD-NL is able to reproduce this response very accurately. This shows that ABCD-NL is indeed an accurate and powerful way to Booleanize non-linear data converters for analysing mixed-signal systems.

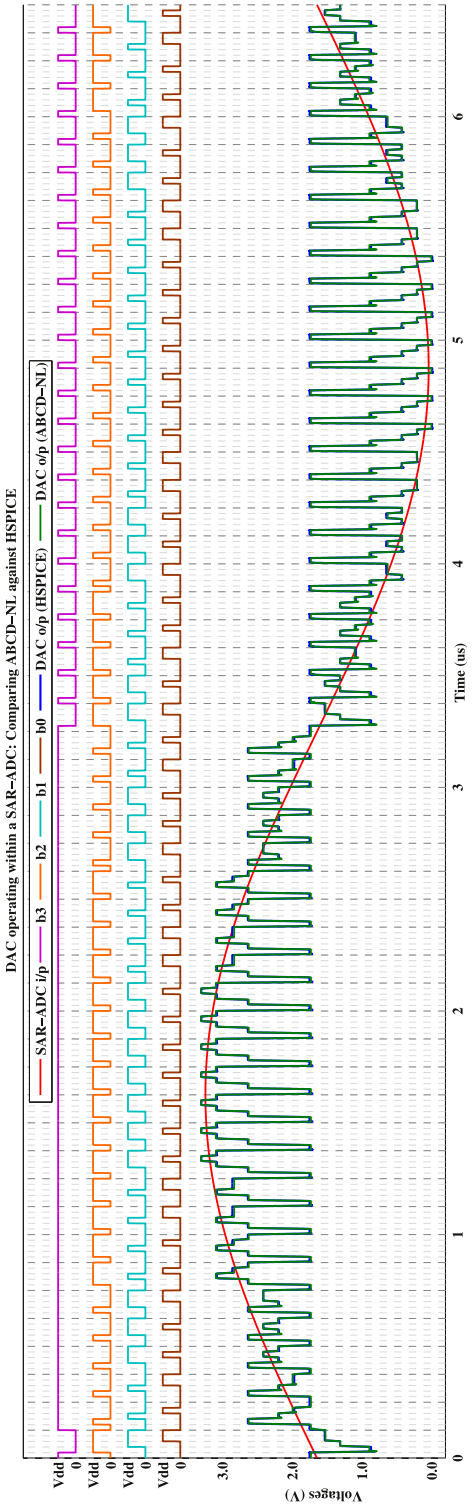


Figure 6.32: ABCD-NL, using a purely Boolean model, is able to accurately capture the dynamics of a D/A converter embedded within a SAR-ADC, across a long time frame encompassing several ADC samples.

6.8 Example: Booleanizing an analog comparator

We now apply ABCD-NL to an analog comparator, another key component in a SAR-ADC. For this demonstration, we shall Booleanize an off-the-shelf comparator (LT1016 from Linear Technology, whose SPICE model is available online [72]), and deploy the resulting Boolean model in a SAR-ADC environment.

Booleanizing a SAR-ADC comparator is a particularly challenging problem because the circuit is highly non-linear, and very sensitive to its (large signal) inputs; for example, a differential input of 1mV elicits a very different response from the system compared to a 2mV (or 1V) differential, in terms of delay, the final steady state solution, *etc.*. This necessitates very fine discretization of the input waveforms, which can in turn result in a very large Boolean model unless domain knowledge is used (see below). Moreover, for each input sample of the SAR-ADC, the comparator usually begins at a large differential (of the order of 1V), and the closed-loop dynamics of the system uses feedback to progressively make this differential smaller, until it is of the order of 1mV. And the Boolean model must capture the dynamics of the comparator, to high accuracy, in spite of this large operating range.

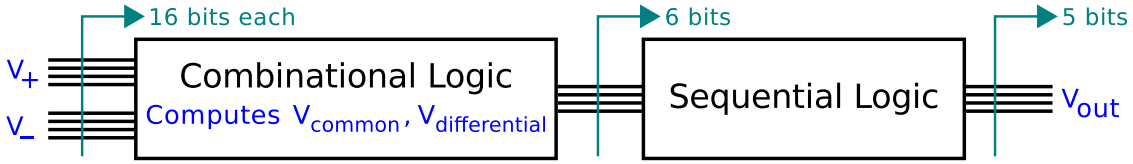


Figure 6.33: ABCD-NL used with domain knowledge to achieve better efficiency. For a comparator, rather than directly operating on the input bits, it is significantly more efficient to first transform the input signals into common mode and differential mode components (using combinational logic), which can then be used to drive ABCD-NL's sequential Boolean model.

As indicated above, to reduce the size of the Boolean model (at the expense of push-button style "automatedness"), we use domain knowledge with ABCD-NL. Fig. 6.33 illustrates this point for the comparator. Because the input has to be discretized very finely (*e.g.*, using 16 bits or more per signal), it is inefficient to apply ABCD-NL directly to the discretized input. Instead, we use a combinational logic unit to compute the *common mode* and *differential* components of the input, which can be discretized relatively coarsely (*e.g.*, using just 6 bits for the differential mode, and 0-2 bits for the common mode). In particular, the differential component is discretized non-uniformly, placing more emphasis on small differentials (because they exert a powerful influence on system dynamics), and a smaller emphasis on large differentials (where the behaviour quickly saturates).

Fig. 6.34 illustrates the results. Part (b) of the figure shows that ABCD-NL, with the efficiency enhancements above, is able to accurately reproduce the SPICE-level analog behaviour of the comparator, for a wide range of input excitations. These excitations are generated as follows: initially, the input differential between the inputs V_+ and V_- is chosen

to be either *large* (1V) or *small* (1mV). This choice has the effect of biasing the comparator either at a strongly polarized bias point, or a weak bias point. Now, a second choice is made: the input differential is suddenly reversed in polarity, using either a small driving strength (1mV differential), or a large driving strength (1V differential). This choice has a significant impact on response time; for example, a strong differential signal starting from a weakly polarized system state evokes a much faster response than a weak differential trying to flip the system from a strongly polarized state. The goal is to design the Boolean model to accurately capture all the different corner cases, and as Fig. 6.34 (b) shows, the Boolean model produced by ABCD-NL achieves this goal (using 5 bits to encode the output waveform).

Also, for verification purposes, it is important to model the comparator's departure from ideal behaviour. An important factor in this context is the DC sensing offset of the comparator. For example, Fig. 6.34 (c) shows a situation where the comparator behaves in a highly unexpected way: even though V_- is always higher than V_+ (*i.e.*, an ideal comparator's output would remain low throughout), the LT1016 actually switches from low to high. Such unexpected behaviour can potentially introduce bit-errors in the context of a SAR-ADC, due to incorrect decisions made by the comparator within the feedback loop. Therefore, while analyzing a SAR-ADC that uses this comparator, it is important to use a comparator model that accurately accounts for such imperfections and shortcomings. And as Fig. 6.34 (c) shows, ABCD-NL's Boolean model does accurately reproduce the behaviour of the comparator (because some of the step input waveforms used by ABCD-NL during the model generation phase triggered such imperfect responses). This is a powerful advantage offered by ABCD-NL – anything that SPICE can predict, the Boolean model can incorporate.

Finally, Fig. 6.34 (d) shows that ABCD-NL accurately reproduces the behaviour of the comparator when it is embedded in a typical SAR-ADC environment. The ADC, and the input to it, are the same as in Fig. 6.32. Over one time period of the input waveform (*i.e.*, 52 ADC samples), the top half of Fig. 6.34 (d) plots the comparator inputs V_+ and V_-. The bottom half of the figure shows that ABCD-NL is able to capture the system's response, almost to SPICE-level accuracy, over this entire time frame.

6.9 Limitations of ABCD-NL

Based on the results of the previous sections, we believe that ABCD-NL can indeed be a powerful AMS modelling approach that facilitates effective, bug-free AMS design. But it is not without limitations, as discussed below.

- **Lack of a rigorous mathematical underpinning.** While we have good empirical evidence and promising results, we still do not have a strong mathematical theory that guarantees the goodness of approximation achieved by ABCD-NL-generated Boolean models for general non-linear AMS systems. Choosing Booleanization parameters (*e.g.*, the number of quantization levels, the clock period for the Boolean model, *etc.*) still requires circuit intuition and perhaps some trial and error, and at this point is more

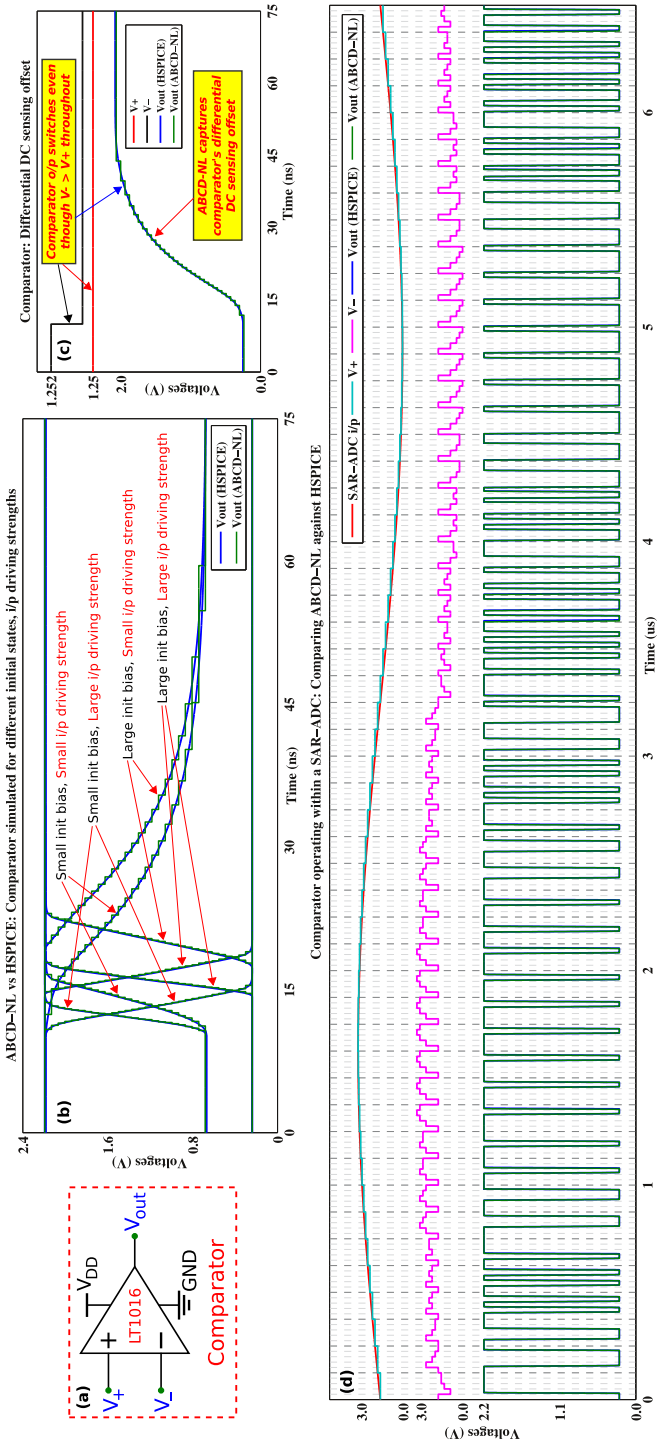


Figure 6.34: ABCD-NL applied to a Linear Technology LT1016 comparator (part (a)). Part (b) shows that ABCD-NL is able to capture the dynamics of the comparator over a wide range of differential input excitations (from 1mV all the way to 1V). Part (c) shows that ABCD-NL can duplicate the SPICE-level dynamics of the comparator even when there is serious departure from ideal behaviour. Part (d) demonstrates that ABCD-NL is well-suited to model the comparator in the context of a SAR-ADC.

art than science. We fear that it may remain so for a long time to come, since the analysis of non-linear DAEs is a very difficult mathematical problem.

- **Boolean models can grow large.** As with other AMS modelling approaches, ABCD-NL-generated Boolean models that capture SPICE-level dynamics to high accuracy can sometimes grow large (especially if a large number of bits are used to discretize the underlying signals). For instance, ABCD-NL works by enumerating all possible discretized circuit input combinations (in order to generate the DC states of the FSM). If the number of input combinations is large (either because the number of circuit inputs is large, or because the circuit inputs are discretized finely), this can take a long time and result in a large Boolean model. Moreover, ABCD-NL carries out transient simulations between every pair of DC states so generated, which can be prohibitively expensive when there are a large number of DC states. While large Boolean models do not necessarily impact simulation speeds (because simulation can usually be done in close to $O(1)$ per time-step, regardless of model size), they can be problematic for applications like AMS verification.

- **Not all continuous systems are Booleanizable via ABCD-NL.** As we mentioned in Section 6.1, ABCD-NL can only be applied to Booleanize reasonably well-behaved continuous systems; for example, ABCD-NL assumes that the system being Booleanized satisfies basic existence and uniqueness properties, features reasonably smooth waveforms (otherwise, most jump heuristics would fail), is robustly stable and non-chaotic, responds with (eventually) constant outputs to constant inputs regardless of initial system state, *etc.*. While most real-world AMS systems do satisfy these properties, there also exist significant exceptions, such as oscillators, that are not Booleanizable via ABCD-NL.

- **New formal verification techniques may be needed.** While existing model checking tools (*e.g.*, ABC, SuperProve, *etc.*) are indeed usable for AMS verification via Booleanization, we must remember that such tools were developed and optimized primarily for digital designs, not Booleanized AMS designs. These verification techniques cannot take advantage of the very special structure underlying ABCD-NL-generated models, namely, the organization of the model into DC states, TRAN paths, and jump heuristic based arcs. Therefore, to formally verify complicated AMS systems comprising many intricate and tightly coupled AMS components in an efficient and scalable way, we may need to develop specialized verification techniques (*e.g.*, "word-level" verification approaches) targeted at ABCD-NL-generated Booleanized AMS models.

- **ABCD-NL-generated models may not guarantee verification safety.** We would like to reiterate that Booleanizing AMS designs (in general, as well as in particular, using ABCD-NL) only results in approximations to SPICE-level behaviour. In many cases, the Booleanization process can be "hardened" (*e.g.*, by introducing non-determinism into either the ABCD-NL algorithm or the resulting Boolean model)

to produce overapproximations that capture the underlying circuit's behaviour conservatively and safely. However, depending on the property to be verified, it may not always be easy to come up with such safe models. Thus, while the Boolean model that is verified may be an excellent approximation to SPICE (and one that is much closer to reality than many hybrid system models), such a model may still not be guaranteed to catch all corner cases, and hence may compromise the soundness and completeness of verification. Unfortunately, this is a generic limitation of most analog modelling approaches that AMS designers have been forced to accept.

6.10 Summary

To summarise, in this chapter, we have developed and demonstrated ABCD-NL, a technique for carrying out the Booleanization of genuinely analog and genuinely non-linear analog AMS components and sub-systems. The Boolean models generated by ABCD-NL are suitable for high-speed simulation and formal verification of AMS designs using existing tools, methodologies, and flows based around either Boolean analysis or hybrid system frameworks. We have applied ABCD-NL to several circuits that are of interest to AMS designers, including charge pumps, signalling/communication sub-systems, delay lines, D/A converters and comparators used in SAR-ADCs, *etc.*.

Software implementation: We have made a preliminary version of ABCD-NL (implemented in Python) available under an open-source license at `http://jr.eecs.berkeley.edu/ABCD`. We would appreciate comments, feedback, and suggestions on this. For more details about how the software is organized, documentation on how to use it to Booleanize AMS systems and compare the resulting models against SPICE, how to import the Booleanized models into formal verification engines like ABC or SuperProve, *etc.*, please see the website above. We do not supply these details here because the software is constantly evolving; therefore, these details, if included in this dissertation, would quickly become out of date.

With this, we conclude the part of this dissertation that was devoted to developing algorithms, tools, and techniques for Booleanization. Under the umbrella of ABCD, we have developed and demonstrated 3 such techniques (in this chapter and in the preceding two chapters). These techniques include DAE2FSM (for Booleanizing "digitalish" systems plagued by analog non-idealities), ABCD-L (for Booleanizing LTI systems), and ABCD-NL (for Booleanizing genuinely analog, genuinely non-linear AMS designs).

Thus far, the focus of this dissertation has largely been on *modelling* AMS components and sub-systems. The whole idea of Booleanization may itself be viewed as a novel way to *model* AMS systems with a view to retaining good accuracy without unduly sacrificing scalability or amenability to analysis.

In the next chapter, we will focus on the kinds of applications made possible by the Boolean models generated above, rather than the model generation itself which has been our chief focus up to this point. For example, we will discuss the implications for high-speed simulation of AMS designs using ABCD-generated models. We will present an example that

involves using ABCD-NL-generated models, in conjunction with ABC [6], for formally verifying a signalling system. We will also explore new kinds of analysis algorithms that involve a mixture of continuous and Boolean models. In this context, we will develop and demonstrate BEE, a Boolean/LTI co-analysis technique for accurately predicting eye diagrams in modern high-speed communication sub-systems that employ coding strategies as part of their design.

Chapter 7

Uses of Boolean models

In many ways, we can say that the chief focus of this dissertation upto this point has been the *accurate modelling* of AMS systems. Indeed, the very process of Booleanization can be thought of as a novel way to model AMS designs – finding ways and means to represent and abstract analog dynamics into Boolean form. In particular, the last three chapters focused entirely on developing a suite of Booleanization techniques for different kinds of AMS systems: DAE2FSM for "digitalish" systems, ABCD-L for LTI systems, and ABCD-NL for genuinely analog, genuinely non-linear systems. These techniques all generated Boolean models of AMS designs.

In this chapter, we will focus on how such Boolean models can be put to use. That is, once we Booleanize an analog/mixed-signal system, in what ways can we analyze the resulting Boolean model to answer meaningful questions about the original system and/or improve its design?

We will cover three important uses for such Boolean models: (1) high speed simulation, (2) formal verification, and (3) custom analyses involving Boolean models to address specific target problems in the AMS design space, such as eye diagram analysis.

7.1 High-speed simulation of AMS designs

As mentioned before in this dissertation (*e.g.*, in Sections 3.8, 5.7, and 6.2), Boolean models have a powerful advantage over continuous or hybrid system models when it comes to simulation. This is because, to simulate a continuous or hybrid system, one needs to solve a system of differential equations. This typically requires matrix solutions, device evaluations, linear multi-step methods, Newton-Raphson iterations, *etc.* [7]. These are all time-consuming operations, with per time-step complexity that is at least linear (often much worse, and always with a large constant in front) in the size of the design being simulated. Therefore, the simulation of continuous systems (*e.g.*, SPICE simulation) and hybrid systems tends to be slow. On the other hand, the simulation of purely Boolean systems can be as simple as a single memory lookup per time-step, with time complexity $O(1)$ per time-step, *i.e.*, independent

of the size of the design being simulated. This, of course, is very attractive. Below, we make these arguments clearer and more precise by describing the high-level flows used to simulate SPICE-level models, hybrid system models, and Boolean models.

First, we would like to clarify what we mean by "simulation". Analog/RF designers are familiar with many different kinds of simulation, including DC analysis and DC sweeping, AC analysis, transient simulation, harmonic balance analysis, *etc.*. However, when *we* say "simulation", we mean "transient simulation", which is stated as follows: given a system to be simulated, and given the initial state of the system (say, at time t_0), and given the inputs to the system over a time interval (say, $[t_0, t_f]$) beginning at t_0, the problem is to predict the outputs of the system over this time interval.

High-level transient simulation flow: The problem above is solved by first breaking up the time interval $[t_0, t_f]$ into small chunks, where each chunk corresponds to a time-step in the simulation. The simulator starts at the first time-step (beginning at time t_0), and then proceeds successively from one time-step to the next, until it reaches the last time-step (ending at time t_f). The system's state (and outputs) at each time-step are calculated from those at previous time-steps, as well as the system's inputs at the current time-step. Finally, the system's outputs over the entire time interval $[t_0, t_f]$ are collected and returned.

7.1.1 SPICE-level simulation

We now describe the the high-level flow used to simulate SPICE-level models (typically, by describing a SPICE-level netlist to a SPICE simulator like HSPICE or Xyce), and the computational considerations involved.

As described in Section 1.2.3, SPICE-level circuits (netlists) are internally represented as DAEs by SPICE simulators. Given a circuit, each device in it contributes a few terms to a DAE, and the SPICE simulator can construct the entire DAE for the circuit by combining together the terms from each device, based on the way the devices are wired together (as described in the netlist). Given the circuit's inputs over time, the unknowns (to be solved for) are the voltages and the currents in the circuit over time, which make up the DAE's state vector (say, $\vec{x}$). The circuit's outputs over time are computed from the solution to the DAE (once the SPICE simulator has successfully solved for the unknowns), and returned.

Initial condition: This is the initial state vector, $\vec{x}(t_0)$, of the DAE at time t_0, corresponding to all the voltages and currents in the circuit at time t_0 (from which the state of each device in the circuit, including charges on each capacitor and currents through each inductor, can be derived). The initial condition can be either specified in the netlist or calculated by the simulator (*e.g.*, by solving for DC voltages and currents at time t_0).

Calculations done at each time-step: At each time-step, the SPICE simulator needs to solve for all the voltages and currents in the circuit (*i.e.*, the DAE state vector $\vec{x}$), using the voltages and currents at previous time-steps, as well as the current inputs. This typically requires constructing and solving a non-linear algebraic equation.[1] To construct this equa-

[1]We note that this equation is *not* the same as the circuit's DAE. This is a purely algebraic equation,

tion, the simulator typically carries out a few computations based on the current inputs to the circuit, as well as the DAE state vectors computed at previous time-steps. The precise formula used for constructing this equation depends on the so called "integration method" used by the simulator. Common choices for this integration method include forward Euler, backward Euler, trapezoidal, GEAR, *etc.* [7].

Once the equation above is constructed, the next step is to solve the equation to obtain the voltages and currents in the circuit at the current time-step. This is done using an iterative process called "Newton-Raphson" (or NR, for short) [7]. The number of NR iterations required to solve the equation differs from time-step to time-step. "Easy" time-steps may only take a few iterations, while "hard" time-steps could take a few tens of iterations. Still harder time-steps may force the NR routine to give up after trying a few hundred iterations; in such cases, the simulator either throws an error, or resorts to more advanced (but computationally more expensive) solution techniques like arc length continuation based homotopy.

At each NR iteration above, the simulator "evaluates" (*i.e.*, calls a few functions implemented by) each device in the circuit. The results of all these device evaluations are then combined into an $A\vec{x} = \vec{b}$ problem, *i.e.*, a matrix vector problem. This is then solved using a technique like Gaussian elimination.

Per time-step complexity: At each time-step, the simulator needs to first construct the non-linear algebraic equation to be solved. This takes time of the order of the size of the circuit's DAE, which is of the order of the number of devices in the circuit. Then, the solution of this equation requires a few NR iterations. Each of these iterations involves constructing and solving an $A\vec{x} = \vec{b}$ problem. The construction of this problem calls each device in the circuit at least once, so its time complexity is at least the number of devices in the circuit. Then, solving the problem using a technique like Gaussian elimination, even when using highly optimized sparse matrix techniques and solvers, takes time at least of the order of the number of devices in the circuit.

Thus, the best case scenario is that, on average, SPICE simulation takes $O(kn)$ time, where k is the average number of NR iterations per time-step, and n is the number of devices in the circuit. If NR fails often, or if the sparse matrix solver does not work as well as expected, this best case scenario no longer holds true, and the complexity may increase significantly (perhaps to quadratic or even cubic in the number of devices in the circuit).

Practical experience: In practice, SPICE is often unacceptably slow for all but the smallest and simplest AMS systems. This is because, as described above, the simulation flow is highly complicated and involves several time-consuming device model evaluations, matrix vector solutions, NR iterations, *etc.*. Even in the best case scenario, there is a large hidden constant before the $O(kn)$ described above. For example, at each time step, SPICE needs to evaluate each device in the circuit a few times. These device evaluations can be very costly, especially when advanced device models like BSIM or PSP are used to model the transistors

whereas the circuit's DAE typically has both differential and algebraic components. Also, this is a per time-step equation, *i.e.*, it changes with each time-step – unlike the circuit's DAE which is fundamental to the circuit and hence does not change with time.

in the circuit. For reference, a single device evaluation of a BSIM model typically requires a few hundred thousand floating point operations. This all contributes to the large constant above, making SPICE-level simulation rather impractical for most real-world AMS designs. However, because SPICE simulations are highly accurate, they can be (and often are) used to characterize small individual components in large AMS designs, for purposes like model order reduction, table-based modelling, Booleanization, *etc.*.

Accuracy: Since the simulation flow in SPICE is based on repeatedly evaluating detailed device models capturing almost all features of the underlying system, it is highly accurate. Indeed, many AMS designers implicitly (and with good reason) trust SPICE, and routinely use the predictions made by SPICE as proxies for how their designs will behave in the real world. Therefore, SPICE simulations are considered the golden standard against which other simulation techniques are compared to gauge their accuracy.

Other important considerations: Here, we would like to reiterate that, outside of a SPICE simulator, a SPICE-level model for a circuit is not of much use. Such a model cannot be directly used for formal verification or other kinds of custom analyses; it is useful for SPICE-level simulation, and SPICE-level simulation only. Depending on the size of the circuit (*i.e.*, the number of devices in it), and how complicated the devices in the circuit are, even such SPICE-level simulation can be prohibitively expensive.

7.1.2 Hybrid system simulation

We now describe the the high-level flow used to simulate hybrid system models, and the computational considerations involved.

Recall (from Section 2.2) that hybrid system models feature hybrid automata that consist of "places", transitions between places, and guard conditions. Each place in a hybrid automaton is typically associated with a DAE, and when the state vector of this DAE meets one or more of several guard conditions, the hybrid automaton transitions to a different place. Below, we describe what the simulation of such a system involves.

Initial condition: This includes the initial place (at time t_0) of the hybrid automaton being simulated, as well as the initial state vector of the DAE at this initial place.

Calculations done at each time-step: At each time-step, a hybrid model simulator needs to first solve for the next state of the DAE at the current place; the DAEs of all places other than the current place can be ignored. As described in the previous section on SPICE simulation, this DAE solution involves constructing and solving a non-linear algebraic equation, via NR.

Having solved for the DAE state, the hybrid model simulator then needs to test whether this state satisfies any guard condition outbound from the current place. If not, the simulator is done for this time-step. But if one or more guard conditions are satisfied, the simulator needs to execute an appropriate place transition, and then load into memory the DAE associated with the new place; this will be the DAE that is simulated in the next time-step.

Per time-step complexity: From the discussion above, in the best case scenario, the average per time-step complexity of hybrid automaton simulation is $O(kn)$, where n is the

average number of DAE state variables (since the DAEs associated with different places can have different sizes), and k is the average number of NR iterations required per time-step.

Practical experience: As described in Chapter 2, most hybrid models used in practice are constructed expressly for formal verification, *not simulation*. Also, as mentioned in Chapter 2, most formal verification techniques do not scale well with the number of continuous variables involved. Therefore, the DAEs that typically feature in practically relevant hybrid automata are not very large. Indeed, we have never seen a hybrid automaton in the literature with more than a handful of DAE state variables. Therefore, the per time-step NR solves above are usually much quicker than typical NR solves in SPICE-level simulations.

Thus, while the techniques used to simulate the DAEs in hybrid models are similar to those used by SPICE, hybrid model simulations are typically much faster than SPICE because the DAEs involved are much smaller. Moreover, in addition to being smaller, these DAEs are much *simpler*. Indeed, more often than not, these DAEs tend to be linear ODEs. For such systems, NR provably converges to the correct solution in a single iteration, involving just a single matrix solution. Therefore, for such systems, the k in the $O(kn)$ above drops to 1, and as a result, the per time-step complexity drops to just $O(n)$. Moreover, as described above, n (the average DAE size in the given hybrid automaton) usually never exceeds a handful (5 or 10). So, for most practical purposes, real-world hybrid automata can be simulated in roughly $O(1)$ per time-step. But of course, the constant in front of this $O(1)$ is not quite negligible, since even a 5×5 or 10×10 matrix solution can add up over a large number of time-steps.

Accuracy: Unfortunately, as described in Chapter 2, hybrid automata that occur in the real world tend to be significantly less accurate than SPICE-level models. Indeed, in most practical applications (see Chapter 2 for examples), hybrid automata are constructed by highly oversimplifying the SPICE-level dynamics of AMS systems. For example, nonlinear dynamics (the source of most non-ideal circuit behaviour and design bugs) is usually completely ignored, in an effort to construct hybrid automata that can be formally verified in reasonable time. Indeed, most hybrid models are far removed from transistor-level reality; therefore, while comparing the results of hybrid model simulation against SPICE simulation, the best that can be hoped for is a qualitative match (as described in Section 3.4), if that. Quantitative comparisons are apt to reveal large discrepancies. This *loss of accuracy* that is characteristic of hybrid system models is often unacceptable to AMS designers.

7.1.3 Booleanized AMS system simulation

We now describe the the high-level flow used to simulate Booleanized AMS models (*e.g.*, the models produced by ABCD), and the computational considerations involved.

At the outset, we note that different kinds of Boolean models have different costs of simulation; there is usually a tradeoff between model size (*i.e.*, "memory") and simulation cost (*i.e.*, "CPU"). Large Boolean models occupy a lot of memory, but they tend to be faster to simulate. Below, we consider the different kinds of Boolean models that ABCD can generate, and we discuss the computational characteristics of simulating each.

Initial condition: For combinational Boolean models, there is no need to supply an initial condition, since these systems are "memoryless" (*i.e.*, their current outputs are instantaneous functions of their current inputs, with no dependence on past history). On the other hand, the simulation of sequential Boolean models requires an initial condition to be specified. If the model is represented as an FSM state transition graph (STG), the initial condition would be the initial FSM state (that the system is in at time t_0). If, on the other hand, the model is represented in Boolean circuit form (*i.e.*, using logic gates and latches,[2] such as a sequential AIG), the initial condition would be the initial (binary) state of all the latches in the circuit.

Calculations done at each time-step: Simulating a combinational system simply involves evaluating the system's outputs (for the given inputs) at each time-step and returning them. This is because the memoryless property of these systems ensures that there is no need to consider the state of the system at previous time-steps. For example, if the Boolean model for such a system is stored as a truth table (*i.e.*, a *hash table* with the system's inputs as keys and the corresponding outputs as values), all that is required is a single memory lookup. On the other hand, if the system is stored in a more compact, circuit-encoded form (such as a BDD or AIG, stored as a directed acyclic graph, or DAG), then evaluating the output involves traversing the circuit DAG.

A sequential system, on the other hand, has a notion of "state". Therefore, simulating such a system requires updating its state at each time-step, in addition to evaluating and returning its outputs. For example, if the Boolean model representing such a system is stored as an FSM STG (*i.e.*, a hash table mapping (current FSM state, input symbol) pairs to (next FSM state, output symbol) pairs), then simulating the system again involves a single memory lookup per time-step. If, on the other hand, the Boolean model is stored in circuit-encoded form (*e.g.*, as a sequential AIG), then the "next state given current state and inputs" function, as well as the "outputs given current state and inputs" function, are both available as DAGs, which are traversed at each time-step to carry out the simulation.

Per time-step complexity: From the discussion above, the per time-step complexity of simulating a Booleanized AMS system can be as efficient as a single memory lookup, which takes $O(1)$ time, *i.e.*, independent of the size of the system being simulated, and also independent of the size of the original analog/AMS system that was Booleanized into this form. In the worst case, the simulation can involve a couple of DAG traversals, taking $O(n)$ time, where n is the size of the DAG being traversed.

Practical experience: In our experience, FSM STG model based Booleanized AMS simulation is easily the fastest amongst the methods considered so far: it handily beats both SPICE and hybrid model simulation. In many cases, the Boolean models produced by ABCD are small enough that they can be stored in FSM STG form; in these cases, simulation can be very fast indeed, with just a single memory lookup per time-step. The comparison is clear: a few million floating point operations per time-step for SPICE, versus a few hundred to a few thousand floating point operations per time-step for hybrid system simulation, versus a sin-

[2]Here, we use the terms "latches" and "flip-flops" interchangeably, following common practice in the verification community.

gle memory lookup per time-step for Booleanized AMS model simulation. Moreover, unlike SPICE-level or hybrid model simulation, writing software to implement transient simulation of Booleanized AMS systems is very straightforward; our implemenation, for example, is not much more complicated than a single short `for` loop.

Some Boolean models, however, are too large (*i.e.*, occupy too much memory) to be stored as FSM STGs; these are typically stored as AIGs (*e.g.*, in AIGER format), or as circuits encoded as BLIF[3] files. The DAGs associated with these models are usually very compact (with sizes that are logarithmic in the size of the underlying FSM), and can be traversed efficiently while simulating these systems. While such simulations are slower than simulations of Booleanized AMS systems stored as FSM STGs, they can still be many times faster than SPICE. Also, more efficient algorithms than vanilla DAG traversal have been developed for simulating Boolean circuits stored in such forms (*e.g.*, the wavefront propagation inspired method described in [73]), and using these may result in even better speedups.

For a detailed discussion and concrete results on the speedups offered by ABCD-L-generated models over SPICE-like simulation, please see Section 5.7. For a briefer discussion on the speedups over SPICE offered by ABCD-NL-generated models, please see Section 5.2.

Also, we would like to point out that, during this work, we did not focus very much on the high-speed AMS simulation problem. Instead, the lion's share of our focus was devoted to the problem of accurately modelling AMS systems via Booleanization, with some attention given to problems like Booleanization-enabled formal verification of AMS designs and the development of custom analysis algorithms for targeted AMS design problems such as eye diagram analysis. Therefore, we have not explored the full extent of the potential of ABCD and Booleanization for high-speed AMS simulation. Our implementation of ABCD-NL, for example, is in Python, an interpreted programming language that is much slower than compiled languages such as C or C++ for many tasks. Even so, we are able to carry out time-domain transient simulations 5x to 30x faster than comparable simulations carried out by commercial SPICE versions (like HSPICE), on the AMS circuit examples of Chapter 6. If our simulation algorithms were rewritten in C or C++, we feel confident that much higher speedups over SPICE (*e.g.*, 100x or higher) will be achievable.

Accuracy: As we have seen from the many examples covered in the last 3 chapters, the Booleanization techniques developed in this dissertation are able to produce FSMs that are indeed able to achieve high levels of accuracy when compared against SPICE. For most (or all) inputs of interest, the waveforms predicted by ABCD-generated Boolean models closely match those predicted by SPICE, both qualitatively and quantitatively (as described in Section 3.4). In particular, we believe that Booleanized AMS model simulation can offer much higher accuracy, as well as better efficiency, compared to simulating typical hybrid system models available for today's AMS designs. Furthermore, the accuracy of ABCD-generated models is also easily *tunable*: one can improve it simply by increasing the fineness of discretization used by the underlying Booleanization techniques.

Therefore, we believe that the Boolean models produced by ABCD are indeed well-suited

[3]Berkeley Logic Interchange Format.

for high-speed AMS simulation, both from an accuracy and an efficiency standpoint. Thus, Booleanization can provide a viable alternative for fast, approximate simulation of AMS designs.

Other important considerations: We would now like to briefly mention two factors that arise in Boolean model simulation.

The first is the time taken to generate the Boolean model. For a fair comparison of simulation runtimes between Booleanized AMS simulation and SPICE, we also need to consider the time taken to generate the Boolean model from the SPICE-level netlist. Indeed, the plots in Fig. 5.20 were generated after taking this into account. Therefore, we believe that, even after taking into account the time taken to Booleanize an AMS system, simulating the Booleanized AMS model can still be many times faster than SPICE in many cases. This is particularly true for long transient runs, or if a large number of simulations need to be carried out. In such cases, since generating the Boolean model is a one-time cost, it is amortized over a large base, which dilutes its impact on overall runtimes.

The second consideration has to do with the clock period of the Boolean model versus the time-step used by SPICE. In transient simulation of Booleanized AMS models, each time-step corresponds to one clock period of the Boolean model being simulated. But in SPICE, the length of a time-step is either specified by the user or determined by the simulator itself. Large time-steps often lead to shorter simulation runtimes, but may compromise accuracy. To ensure fair comparison, all the speedup figures quoted in this dissertation were obtained by setting the SPICE time-step equal to the Boolean model clock period. A further complication is that SPICE can dynamically change its time-step while it runs, using methods like local truncation error (LTE) based time-step control: when the circuit's state is evolving rapidly, SPICE can use a small time-step, but when the voltages and currents in the circuit are not changing much, it can take much longer time-steps. This kind of optimization is difficult to implement while simulating a general Boolean model. However, we believe that the special Boolean model structures underlying ABCD-L and ABCD-NL lend themselves to this kind of optimization. We have not implemented such optimizations yet, but we do have some ideas: we believe that techniques like τ-leaping [74, 75] can be used to achieve "time-step control" in ABCD-generated Boolean model simulations, without unduly compromising accuracy.

7.2 Formal verification of AMS designs via Booleanization

We believe that *formal verification* of AMS designs is another important use for Booleanization in general and ABCD-generated models in particular. Indeed, there are several off-the-shelf tools and techniques already available for analyzing Boolean models and formally proving properties about them. The entire field of computer aided verification and model checking is devoted to this problem: this is an active area of research where impressive progress has been made over the last few decades. Therefore, Booleanizing AMS systems

enables us to immediately bring to bear the state-of-the-art tools and techniques from this vast body of work to tackle the AMS verification problem head-on. This is made easier by the fact that ABCD-generated models are readily exportable to formats that most existing Boolean verification tools (such as ABC [6]) already accept as input (*e.g.*, PLA, BLIF, AIG, *etc.*). We now present examples of this "verification flow" applied to real-world AMS designs.

Of course, as we remarked before, the main focus of this dissertation is the *accurate modelling of AMS components for verification*, as opposed to the verification itself. But for completeness, we now present two AMS verification examples. In the first example (Section 7.2.1), we Booleanize a Schmitt trigger circuit by hand starting from first principles, and then formally verify important properties of a system containing this Schmitt trigger as a component. In the next example (Section 7.2.2), we Booleanize an AMS system using ABCD-NL, import the resulting Boolean model into a verification engine (ABC, [6]), and carry out formal property checking of the model against an AMS design relevant specification.

7.2.1 Formally verifying a Schmitt trigger system

Upto this point, we have focused on developing automated Booleanization techniques like ABCD-L and ABCD-NL. But in this section, we will Booleanize a Schmitt trigger circuit manually before formally verifying a system containing it.

The main advantage of this is simplicity: we produce a compact Boolean model that is easy to understand and straightforward to specify in typical hardware description languages like Verilog or Chisel [76]. Another advantage is that we leverage the power of nondeterminism to produce a Boolean model that is "safe and sound" for verification purposes. Finally, the "patterns" that we follow here for Booleanizing our system can be adapted and applied to other kinds of AMS systems as well.

This section therefore enables a reader who is new to Booleanization to become acquainted with some basic guidelines for quickly capturing the most important aspects of an AMS system's behaviour in Boolean form, without having to understand/implement more complicated techniques like ABCD-L or ABCD-NL. By design, this section is to a large extent self-contained: one starts with a SPICE netlist for a Schmitt trigger circuit, Booleanizes it by hand following simple manual processes, and then verifies the system using state-of-the-art tools like ABC and SuperProve [6, 10, 11].

7.2.1.1 Booleanizing a Schmitt trigger by hand

We now Booleanize a Schmitt trigger circuit by hand. Within this example, we point out how the underlying techniques can be adapted and applied to other kinds of AMS systems as well.

Fig. 7.1 depicts a transistor-level schematic of a CMOS Schmitt trigger. This circuit has 1 input (V_{in}) and 1 output (V_{out}). The input is typically an analog waveform, whereas the output is (ideally) a digital waveform.

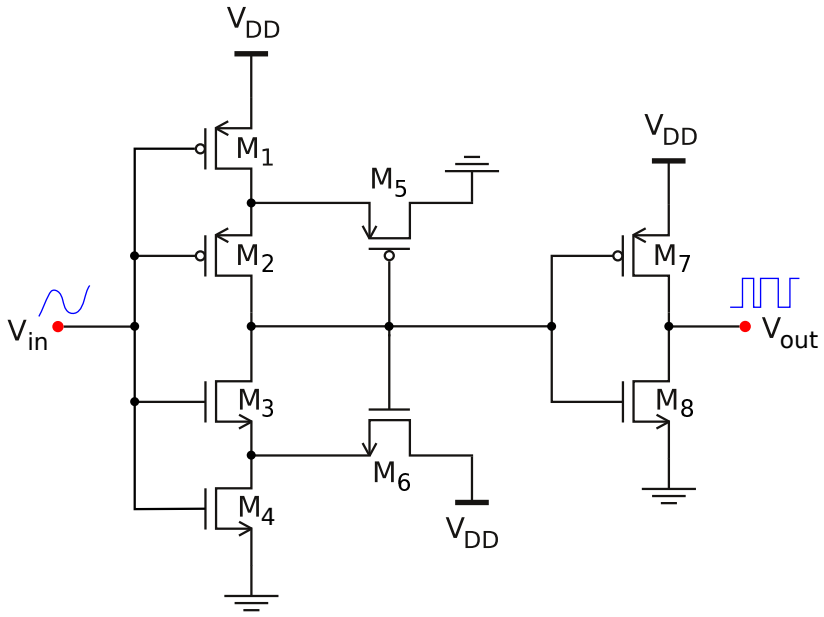


Figure 7.1: Transistor-level circuit for a CMOS Schmitt trigger.

Ideally, the circuit functions as follows: it has 2 *states*, 0 and 1. When the circuit is in state 0 (with output at logic 0), it waits for the input waveform V_{in} to rise above a *high threshold* $V_{th,hi}$, at which time it moves to state 1 (with output at logic 1). Similarly, when the circuit is in state 1, it waits for the input waveform to fall below a low threshold $V_{th,lo}$, at which time it switches its state and output back to 0. A detailed description of how the circuit in Fig. 7.1 works to achieve this functionality can be found in [77].

To Booleanize the circuit above, we designed it (in 90nm CMOS, using BSIM4 transistors) and simulated its behaviour using HSPICE. These simulations helped us determine that the low threshold for the circuit was about 0.623V, while the high threshold was about 1.171V.

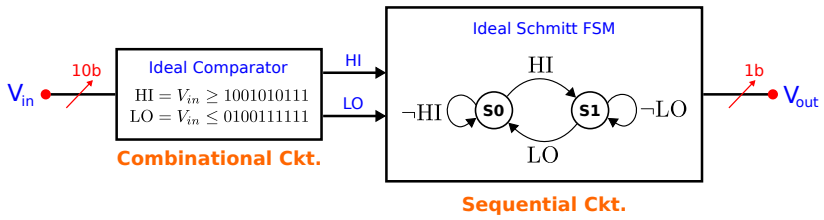


Figure 7.2: Boolean model for an ideal Schmitt trigger circuit.

With this information, it is straightforward to Booleanize the circuit by hand, assuming "ideal" Schmitt trigger behaviour. Fig. 7.2 depicts such a Boolean model, where the continuous input waveform V_{in} (which is assumed to range from 0V to 2V) is discretized using 10 bits. As the figure shows, the ideal Boolean model consists of a simple combinational

Boolean comparator circuit, whose outputs drive a sequential circuit that implements an ideal Schmitt trigger state machine. Just by inspection, one concludes that this is a reasonable Booleanization of an ideal Schmitt trigger. This is an important aspect of Booleanizing systems by hand that generalizes well to many kinds of AMS systems: once the end-to-end functionality of an AMS system is known, it is usually not very hard to come up with a straightforward Booleanization of it, *assuming ideal behaviour*, after discretizing all relevant analog quantities.

A typical real-world Schmitt trigger implementation, however, features many non-idealities not captured by the ideal Boolean model above. To name just three:

1. Uncertain thresholds. A Schmitt trigger's low and high threshold voltage values are never fixed and constant; they typically vary around their nominal values depending on circuit parameters, as well as operating environment variables such as ambient temperature.

2. Hold time considerations. A Schmitt trigger is usually not guaranteed to switch its state unless the corresponding threshold is breached by the input waveform for a minimum amount of time, known as the hold time.

3. State transition delays. A Schmitt trigger does not usually switch states instantly; there is often a delay (due to parasitic capacitances in the circuit) between the input waveform breaching a threshold and the Schmitt trigger switching state.

Building a Boolean model that accounts for such non-idealities takes a little more work. For this, we have several "manual Booleanization patterns". Fig. 7.3 shows a Boolean model, built using these patterns, that nicely captures the non-idealities above.

To capture uncertainties around the Schmitt thresholds, the Boolean model of Fig. 7.3 contains two simple combinational circuits: a "low threshold calculator" and a "high threshold calculator". These blocks add some non-deterministic uncertainty (the extent of which is determined by running SPICE simulations with conservative assumptions on parameter variability) to the low and high thresholds of the Schmitt trigger. Following common practice in digital verification, such non-determinism is modelled by having these blocks accept a few input bits (denoted `ND`) that can take arbitrary values. This idea of introducing non-determinism to model variability and uncertainty around nominal values is a useful pattern that applies to several AMS systems beyond Schmitt triggers.

Hold time constraints in the Boolean model are captured using blocks called "hold time enforcers" (Fig. 7.3). These are shift-register based state machines. The essence of a hold time constraint is that the circuit is guaranteed to respond to an input change only if the input *stays* at its new value for a period of time. To model this, the state machines within the hold time enforcers of Fig. 7.3 have several red states (where no response occurs) and a green state (where a response occurs). These state machines are constructed in such a way that the green state is reached if and only if all hold time constraints are satisfied. For example, if a hold time duration is 3ns (as determined by SPICE simulations) and our

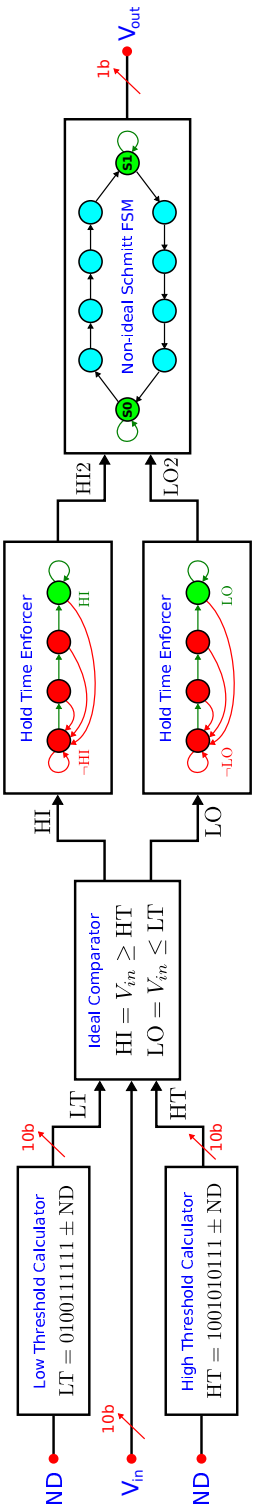


Figure 7.3: Boolean model for a non-ideal Schmitt trigger, taking into account undertain threshold values, hold time considerations, and state transition delays.

Boolean model is clocked at 1ns, we ensure that the state machine passes through 3 red states before reaching the green state. This idea of introducing "obstacle" Boolean model states that need to be traversed through before an action happens is an important pattern that can be applied to Booleanize other AMS systems/behaviours as well. Also, to be conservative, the Boolean model for a hold time enforcer can be non-deterministic: while a response may not be "guaranteed" until a hold time constraint is satisfied, one can use non-determinism to create an execution path where the response still happens. In verification terminology, we say that such a Boolean model is an "overapproximation" of the given circuit, which often adds a margin of safety to the verification flow, at the expense of over-pessimism.

Finally, we would like to model the Schmitt trigger's state transition delays. To this end, Fig. 7.3 features a non-ideal Schmitt FSM that adds delays to the ideal Schmitt FSM of Fig. 7.2. This is again done by introducing intermediate states that the state machine is forced to traverse through before an action happens. For example, the FSM in Fig. 7.3 cannot directly jump from state 0 to 1 or vice-versa: it has to go through several intermediate states (shown in blue) first. Since each intermediate state introduces a delay of one clock cycle, this provides fine grained control for modelling any circuit delay predicted by SPICE; ABCD-L and ABCD-NL use this pattern heavily as well. As before, one can also overapproximate the system via non-determinism, to obtain a wide range of possible conservative delay behaviours.

7.2.1.2 Formally verifying an open loop Schmitt trigger system

Having Booleanized a Schmitt trigger taking non-ideal behaviour such as uncertain thresholds, hold time constraints, and state transition delays into account, we now formally verify a system that contains this Schmitt trigger as a component (Fig. 7.4). This system contains an analog low pass filter (LPF, *e.g.*, an RC filter) followed by a Schmitt trigger. The system's input is a digital signal that bounces frequently between logic levels 0 and 1. This could come, for example, from a highly sensitive photodetector. The LPF (which has a large time constant) produces an analog waveform that rises and falls slowly compared to the rapidly switching input signal. This waveform is then fed to a Schmitt trigger, which switches between 0 and 1 as and when the waveform breaches its low/high thresholds (subject to the non-idealities above). The result is a digital output signal that does not bounce between 0 and 1 as easily as the input (Fig. 7.4); this is practically useful because it cuts out unnecessary noise and fluctuations from the input, providing a steady and reliable signal for downstream applications.

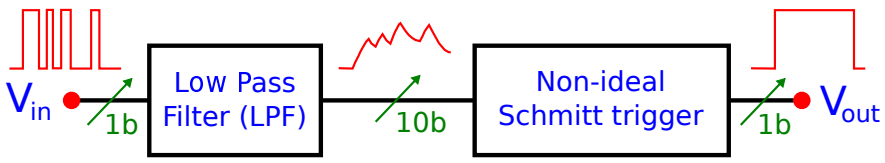


Figure 7.4: The open loop Schmitt trigger system that we formally verify.

Our goal is to prove formally that the system above functions as intended, despite the non-idealities exhibited by the Schmitt trigger. So we Booleanized the system: this involved Booleanizing the LPF (using a technique very similar to ABCD-L), and then composing the LPF Boolean model with the Schmitt trigger Boolean model from the previous subsection. We were then able to use ABC and SuperProve [6, 10, 11] to formally prove the following properties of the system:

1. The system is guaranteed to produce a 1 (0) at the output if any 21 of the last 24 inputs are 1 (0). Thus, if the input waveform remains predominantly at 1 (0), the output waveform is guaranteed to remain steady at 1 (0) even in the presence of an occasional glitch or two, which is exactly what the system was designed for.

2. If the last 18 inputs are 1 (0), the current output is guaranteed to be a 1 (0). This shows that regardless of past history or initial conditions, if the input waveform settles to a 1 (0), the output waveform will also eventually settle to a 1 (0).

3. Finally, if the system's output switches from 0 (1) to 1 (0), it is guaranteed to stay at this new value for at least 4 clock cycles (about 4ns) before it switches again. This speaks to the steadiness and stability of the output; even when the input fluctuates rapidly, the output is guaranteed to not fluctuate as much.

Formally proving the properties above using a conservative Schmitt trigger Booleanization gave us confidence that the system was indeed designed correctly. Also, none of the properties above took more than 10s to prove using ABC/SuperProve running on a modern laptop. Thus, Booleanization can indeed offer a fast and powerful way to formally verify AMS systems.

7.2.2 Formally verifying an AMS signalling system

Figure 7.5: Schematic of a system that was formally verified using a combination of ABCD-NL and ABC. The inverters were designed in a 22nm CMOS process, using BSIM4 models. The channel was modelled as a long RC chain.

Fig. 7.5 depicts the system that we formally verify in this section. It follows the same pattern as the systems shown in Figs. 6.1 and 6.26. As we mentioned earlier, such systems play an important rôle in SI applications, where it is important to determine, and *formally verify*, the throughput of the system. We have used 22nm BSIM4 models for each transistor in the system, and an analog channel that consists of several RC units chained together.

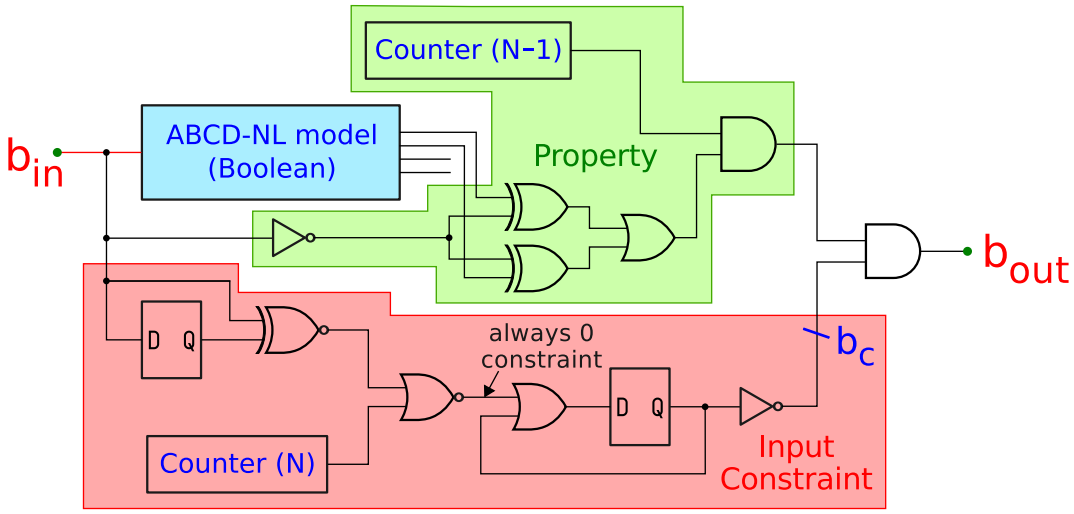


Figure 7.6: Encoding the throughput property, along with constraints on the input, in a Boolean form so as to formally verify the ABCD-NL model using ABC [6].

Fig. 7.6 shows the verification flow that we used. As the figure shows, the Boolean circuit that is verified consists of three parts, (1) the ABCD-NL Boolean model, (2) the Boolean logic that encodes the property to be checked, and (3) some Boolean logic to encode constraints on the inputs that can be applied to the AMS system. The circuit of Fig. 7.6 is constructed in such a way that the given AMS design fails to meet its throughput specification if and only if the bit b_{out} can somehow be asserted to 1 by choosing an appropriate sequence of bits applied at b_{in}.

The constraint on the input is that it can change only once per N clock periods of the ABCD-NL model. The time period of ABCD-NL's sequential Boolean model is 10ps (see Algorithm 1). The input constraint is modelled using a counter that outputs a 1 every N clock cycles (Fig. 7.6). If this constraint is violated, the bit marked b_c immediately becomes 0 and stays there forever, which makes it impossible to assert b_{out} to 1. This ensures that any counter-example returned by ABC would satisfy the input constraint.

The throughput property to be checked is that, given the above constraint on the input, the output should always reach an acceptable state before N clock cycles (*i.e.*, before the input can change). This acceptable state is defined as being $\geq$ 0.8V for a 1, and being $\leq$ 0.2V for a 0.

Clearly, there is an N_0 such that the above throughput property will fail for all $N \leq N_0$. We can use ABC to quickly zero in on N_0, by incorporating ABC-based verification within a binary search loop. In this way, we were able to determine that $N_0 = 38$. This translates to a throughput of approximately 2.56Gbps. Furthermore, we were also able to confirm, using HSPICE, that for $N = 38$, the counter-example returned by ABC is a valid one in the analog domain (Fig. 7.7). Therefore, the throughput bound obtained is tight and meaningful. Thus, ABCD-NL is a powerful and capable modelling technique that can be used for AMS

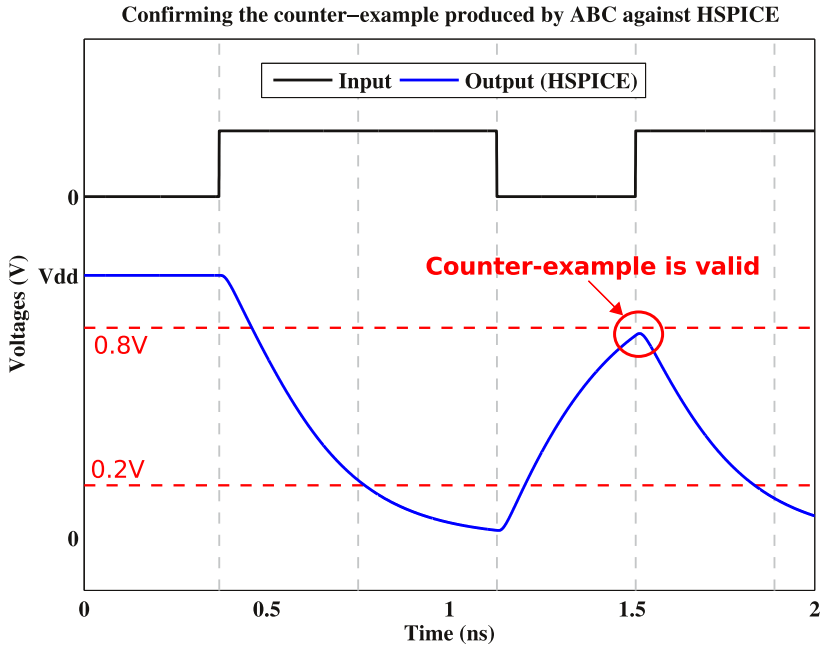


Figure 7.7: Checking that the counter-example returned by ABCD-NL + ABC is valid in the analog domain.

verification.

7.3 Boolean/LTI co-analysis using the Berkeley Eye Estimator

In addition to applying off-the-shelf tools and techniques for analyzing our Boolean models, another promising alternative is to develop our own *custom* tools, techniques, algorithms, and heuristics for specific problems that arise in AMS design.

We now present such a "custom" analysis technique, dubbed the Berkeley Eye Estimator (BEE), which combines Boolean analysis with LTI system analysis (*i.e.*, Boolean/LTI co-analysis) to accurately predict eye diagrams in multi-Gbps I/O links in the presence of correlated bitstreams and coding strategies. This has several applications in signal integrity (SI) analysis and the design of high-speed communication sub-systems. This shows that Boolean models are not only useful for high-speed simulation and formal AMS verification: they can also be the basis around which custom analysis techniques and procedures are designed to solve longstanding interesting problems in AMS design that impact specific kinds of AMS circuits and components. For example, as mentioned above, BEE combines elements of Boolean system analysis with LTI system analysis, so it may be viewed as a custom Boolean/LTI co-analysis method developed to address a specific class of challenging

problems in AMS design, namely, the design of high-speed I/O links to stringent bit error rate (BER) specifications.

7.3.1 The need for BEE

Modern high-speed links and I/O sub-systems often employ sophisticated coding strategies to boost error resilience and achieve multi-Gb/s throughput. The end-to-end analysis of such systems, which involves accurate prediction of worst-case and stochastic eye diagrams, is a challenging problem. Existing techniques such as Peak Distortion Analysis (PDA) typically predict *overly pessimistic* eye diagrams because they do not take into account the coding strategies employed. Monte-Carlo methods, on the other hand, often predict *overly optimistic* eye diagrams (because they do not exhaustively cover all corner cases), and they are also very time-consuming. Thus, an alternative eye diagram estimation technique is needed that is both accurate (*i.e.*, involves neither undue optimism nor excessive pessimism) and efficient (*i.e.*, works well for modern large-scale high-speed I/O links and communication sub-systems that employ coding strategies). This is precisely what BEE attempts to be.

BEE is a computational technique that applies dynamic programming algorithms to predict realistic worst-case and stochastic eye diagrams in modern high-speed links and I/O sub-systems – with neither excessive pessimism nor undue optimism. BEE is able to fully and correctly take into account many features underlying modern communication systems, including arbitrary high-level Boolean transmit-side coding schemes and strategies, as well as various low-level analog non-idealities introduced by the underlying channel(s), such as inter-symbol interference (ISI) and crosstalk, asymmetric rise/fall times, jitter, parameter variability, *etc.*. Furthermore, BEE accurately captures the fact that different received bits typically have widely different eye diagrams when a channel is driven by correlated bitstreams generated by coding strategies. We demonstrate BEE on links involving (7,4)-Hamming, 8b/10b SERDES, and Reed-Solomon encoders, featuring channels that give rise to multiple reflections, dispersion, loss, and overshoot/undershoot. We also apply BEE to high-speed communication sub-systems that make use of pre-emphasis and de-emphasis strategies to improve signal integrity. BEE successfully predicts actual worst case eye openings in all these real-world systems, which can be twice as large as the eye openings predicted by overly pessimistic methods like PDA. Also, BEE is an order of magnitude faster (and much more reliable) than Monte-Carlo based eye estimation methods. Since the systems analyzed by BEE consist of encoders and decoders (which are purely Boolean in nature), as well as analog LTI channels (which are purely analog in nature), BEE is what we call a Boolean/LTI co-analysis technique. Therefore, BEE illustrates yet another use for Boolean models of AMS sub-systems: they may be combined with continuous models of other AMS components to carry out custom analysis of a larger system to answer important AMS design relevant questions.

We now provide some more motivation into why we developed BEE.

High-speed signalling/communication links are becoming increasingly prominent in today's cutting-edge mobile chipsets, SoCs, and other high-performance hardware – including

supercomputers, network switches used in data warehouses, high-frequency trading systems, and so on. Such links now play a significant rôle in determining key system-level performance metrics such as speed, power consumption, and reliability [61, 78–80]. Over the past decade, the throughput of high-speed links has improved dramatically, from a few Mb/s to multi-Gb/s. This improvement is due to sophisticated error-resilient communication techniques that rely on transmit pre-emphasis/de-emphasis, new coding and modulation schemes, advanced equalization methods to compensate for inter-symbol interference (ISI), improved PLLs, DLLs, and clock and data recovery (CDR) solutions, advances in transceiver/link design with improved noise/jitter mitigation, *etc.* [61, 78–82].

However, the additional complexity accompanying the above advances has made system design and optimization vastly more challenging, and in many situations, existing computational solutions/design tools are no longer adequate [61, 78, 79]. For example, a key task is to predict the *eye opening* of a high-speed link [61, 78]; this has important implications for noise margins and end-to-end BER analysis, choice of encoder/decoder architecture, design-space exploration for the receiver's front-end, *etc.*.

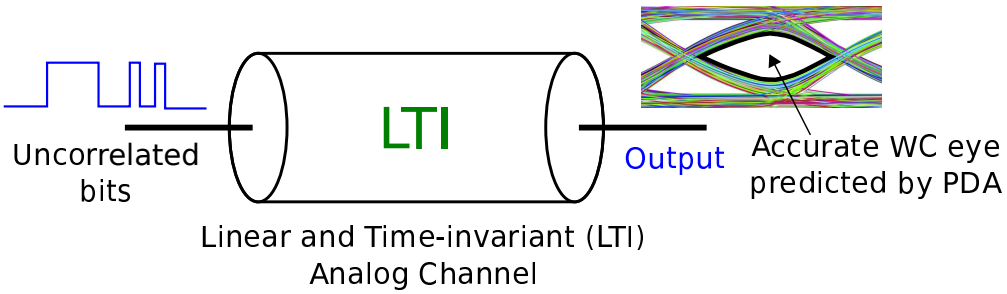


Figure 7.8: Worst case eye diagram computation using PDA.

As depicted in Fig. 7.8, worst-case eye diagrams are often estimated today using Peak Distortion Analysis (PDA) [78, 79, 83], which relies centrally on the assumption that all possible bit sequences are allowed across the channel. However, in modern high-speed links, it is often *the norm* to apply coding strategies [61] to the bits being transmitted, *e.g.*, 8b/10b encoding, error-correcting codes such as Hamming codes, *etc.*. Such coding schemes are designed to improve error resilience by incorporating redundancy in the transmit bits. This significantly restricts the bit sequences allowed across the channel, and as a result, the transmitted bits become tightly correlated. This makes eye prediction by PDA overly pessimistic, which can in turn lead to link over-design, increased design/simulation/debugging costs, sub-optimal link operation, *etc.*. PDA also misses subtler effects, such as the fact that different received bits can have widely different eye openings in the presence of coding strategies (see Section 7.3.2.3).

Due to the above limitations of PDA, high-speed link engineers and communication system architects typically resort to Monte-Carlo simulation to estimate eye openings. However, exhaustive enumeration of all relevant bit sequences is often computationally infeasible, so

only a random subset of bit sequences is used for Monte-Carlo simulation. This frequently misses important bit patterns and corner cases and often predicts *overly optimistic* worst case eyes, especially if the system is being designed for very low BERs (*e.g.*, 10^{-12}). Such overoptimism can lead to serious problems downstream, *e.g.*, increased system-level design/debug time, underperformance requiring expensive post-silicon fixes, *etc.*.

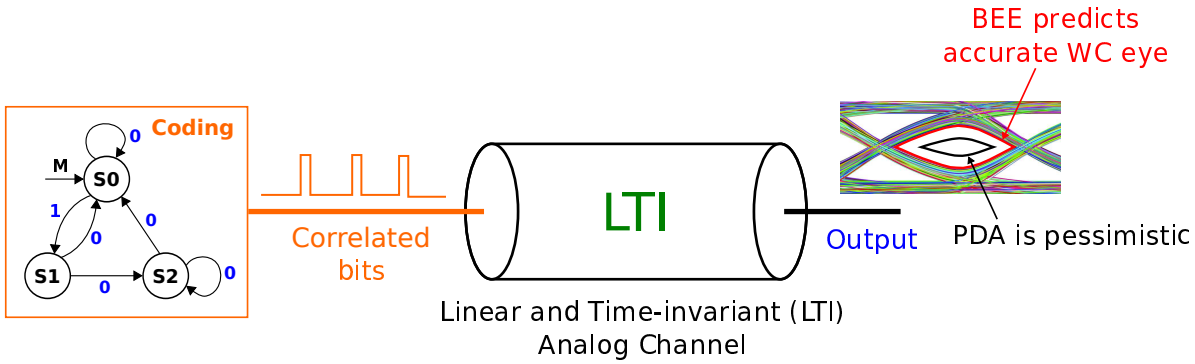


Figure 7.9: Worst case eye diagram computation using BEE.

So we developed BEE, a technique that can efficiently find worst-case and probabilistic eyes of high-speed links with restricted or correlated bitstreams due to coding or other mechanisms. As Fig. 7.9 shows, BEE takes a description of the restriction/coding mechanism in the most general form, *i.e.*, an FSM [12], as well as a description of the (linear) communication channel. It then analyses them together to find the *precise worst case eye*, with neither pessimism nor optimism. Typically, this eye opening is significantly bigger than the pessimistic eye predicted by PDA. If the underlying system is probabilistic (*e.g.*, due to jitter, parameter variability, stochasticity in the coding mechanism, *etc.*), BEE can also predict important properties of the *probability distribution of the eye*, *e.g.*, mean, variance, and higher-order moments.

The core idea behind BEE is to label each state in the underlying FSM with a realistic worst-case scenario, *i.e.*, a sequence of bits terminating at the given state with the highest cumulative cost due to ISI, jitter, *etc.*. At each iteration, the bit sequence associated with each state is expanded by one bit, and this bit is calculated from previously computed worst case sequences via an efficient dynamic programming [84] method. The iterations are continued until the lengths of the computed bit sequences exceed the memory of the underlying channel.[4] At this point, the computed bit sequences are provably the worst-case scenarios for the link, and the corresponding worst-case eye openings are returned. These

[4]The "memory of a channel" is loosely defined as the amount of time it takes for a bit sent over the channel to die down completely, for all practical purposes. In other words, this is the amount of time that must elapse after a bit is sent over the channel, before the effects of sending the bit become virtually undetectable (*i.e.*, the channel "forgets" that the bit was ever sent). While this working definition is not enough to precisely assign a value to the memory of a given channel, it allows one to arrive at a conservative upper bound for it, which is all that BEE requires.

eye openings are actual worst cases; they involve neither excessive pessimism nor undue optimism, unlike PDA or Monte-Carlo based approaches. Further, BEE's time complexity is linear in the size of the given FSM.

BEE provides link design capabilities that are far ahead of any existing technique that we are aware of. It can quickly identify the exact shape of the worse case eye, a capability that fits well with corner-based design methodologies. Also, BEE can identify regions of the eye diagram that correspond to a given probability of occurrence, depicting them graphically to provide immediate visual intuition. Further, BEE correctly captures the fact that different bits of a correlated bitstream can have widely different eye openings, a feature needed for accurate BER estimates. Also, BEE is orders of magnitude faster (and far more reliable in predicting actual worst-case scenarios/probabilities) than Monte-Carlo simulation, making it well suited for practical industrial designs. BEE correctly accounts for ISI, crosstalk, reflections/loss, overshoot/undershoot, dispersion, jitter, and parameter variability – all of which need to be taken into account to make a high-speed link robust in manufacturing practice. BEE's modelling framework is general and widely applicable: virtually any encoder or error correcting scheme that can be implemented digitally (whether using combinational or sequential logic) can be expressed as an FSM, while the linear channel can be specified using SPICE netlists, differential equations, Fourier/Laplace domain transfer functions, measured data, S-parameters, *etc.*. We expect that replacing PDA's pessimistic eyes with BEE's accurate ones during cutting-edge industrial link design will have immediate wide-ranging impact; for example, the adoption of simpler, lower-power encoders with no loss of performance.

We demonstrate BEE on four different types of links. The first (Section 7.3.3) employs a well-known error correcting code, the $(7, 4)$-Hamming code, in the FSM; the channel is modelled as an RLGC chain (a commonly used model for an I/O link/interconnect). The second system (Section 7.3.4) employs an 8b/10b SERDES encoder – a very effective and widely used encoding method employed in several modern standards, including Gigabit Ethernet, IEEE 1394b, PCI Express, SATA and SAS, USB 3.0, *etc.* [85]. The channel here is a weighted sum of smoothed delays, representing a network of heterogenous transmission lines with reflections, or a wireless channel with multiple obstructions and reflectors. The third (Section 7.3.5) is a link that implements a Reed-Solomon encoder, and the fourth and final system (Section 7.3.6) employs a pre-emphasis/de-emphasis based strategy for signal integrity. In all these cases, we demonstrate that PDA makes overly pessimistic worse-case eye predictions, whereas BEE is accurate and exact (as verified against Monte-Carlo). Indeed, the correct eye opening predicted by BEE can sometimes be twice as large as the pessimistic opening predicted by PDA, translating to an exponentially lower BER. Indeed, in some cases, BEE is able to prove the existence of an eye and to precisely compute it, whereas PDA fails, claiming that no eye opening exists. We also demonstrate how BEE accurately takes into account clock jitter and parameter variability in the channel.

7.3.2 Core techniques and eye estimation algorithms underlying BEE

In this section, we first describe some key concepts such as how PDA works, how FSMs can be used to model correlated bits, *etc.*. Then we present the core techniques and algorithms underlying BEE.

7.3.2.1 Peak Distortion Analysis for LTI channels driven by uncorrelated bits

As we mentioned in Section 7.3.1, PDA is a technique for determining WC eye diagrams at the output of a Linear Time Invariant (LTI) channel, when the channel input is a sequence of uncorrelated bits (*i.e.*, no coding strategy is used). We now describe how PDA works.

Given an LTI channel, let $h(t, T)$ denote the channel's response to the following pulse input $u(t, T)$:

$$u(t,T) = \begin{cases} 1 & \text{if } t \in (0,\ T], \text{ and} \\ 0 & \text{otherwise,} \end{cases} \tag{7.1}$$

where t denotes time and T is the pulse width.

Suppose we apply a sequence of bits $\{b_i\}_{i=-\infty}^{+\infty}$ to the above system as input, with each bit being active for a period T. The corresponding input waveform $x_b(t, T)$ is given by the following:

$$x_b(t,T) = \sum_{k=-\infty}^{+\infty} b_k\ u(t - kT, T). \tag{7.2}$$

The channel's response $y_b(t, T)$ to the above input is then given by the following:

$$y_b(t,T) = \sum_{k=-\infty}^{+\infty} b_k\ h(t - kT, T). \tag{7.3}$$

The key idea behind PDA is that we can formulate the worst case (WC) eye computation at time Δ as a pair of optimization problems on a truncated version of the output $y_b(t, T)$, as follows:

$$\begin{aligned} \text{WC0}\ (\Delta,\ T) \quad = \quad & \max_{\{b_i\}} \quad \sum_{k=-M}^{\lfloor \Delta/T \rfloor} b_k\ h(\Delta - kT,\ T), \\ & \text{subject to} \quad b_0 = 0, \text{ and} \end{aligned} \tag{7.4}$$

$$\begin{aligned} \text{WC1}\ (\Delta,\ T) \quad = \quad & \min_{\{b_i\}} \quad \sum_{k=-M}^{\lfloor \Delta/T \rfloor} b_k\ h(\Delta - kT,\ T), \\ & \text{subject to} \quad b_0 = 1. \end{aligned} \tag{7.5}$$

Here, M is a suitably chosen large integer that exceeds the memory of the channel. The crux of PDA is that the solutions to the above optimization problems are straightforward. The intuition is that, simply depending on the sign of the $h(.)$ term, we decide whether we want the corresponding bit to be a 0 or a 1. For example, if we are trying to maximize the output (*i.e.*, the WC0 problem), and we find that $h(\Delta - kT, T)$ is positive for some k, we choose b_k to be 1, and vice versa. In this way, we obtain two optimal solutions, WC0 and WC1, for each Δ. By iterating over Δ, these solutions trace the WC0 (lower half) and WC1 (upper half) of the required worst case eye opening.

The above, of course, was possible only because the bits $\{b_k\}$ were all uncorrelated, and could be set to 0 or 1 independently of one another. On the other hand, if the bits $\{b_k\}$ are correlated (*e.g.*, due to a coding scheme), the entire method breaks down and more powerful techniques (discussed below) are needed.

7.3.2.2 FSMs as a way to model fully-general coding schemes

Our chief concern with BEE is the computation of eye openings in the presence of coding strategies/bit correlations. Therefore, we need a general way to specify these coding schemes/bit correlations. In this context, FSMs are powerful descriptors that enable us to represent virtually any kind of bit correlation/coding scheme in a fast, convenient, and fully general way.

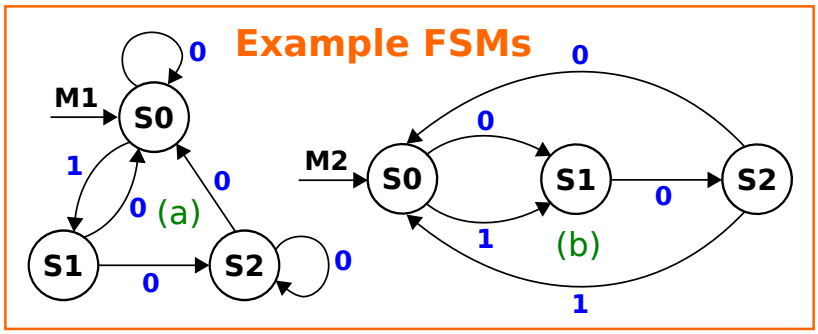


Figure 7.10: Example FSMs representing correlated bits/coding strategies: (a) an FSM that transmits no two consecutive 1s, and (b) an FSM that transmits a 0 every third bit.

Briefly, an FSM is a mathematical abstraction that consists of finitely many *discrete states* (*e.g.*, see Fig. 7.10). Each FSM represents an automaton, *i.e.*, a machine that behaves according to a pre-defined logic. At the beginning, the FSM is in a *start state*. For instance, the FSMs of Fig. 7.10 each have a start state labelled $S0$. FSMs also have arcs (or edges) between states annotated with *output bits*, as Fig. 7.10 shows. The FSM operates in discrete time (*e.g.*, at every uptick of a periodic clock signal). At each time point, the FSM *transitions* from its current state (along one of the arcs directed outward from its current state), reaching a new state (wherever the arc leads to). During this time, the machine's *output* is the bit along the arc that was just followed. For example, in Fig. 7.10 (a), if the FSM is in state

$S0$, it could transition to either $S1$ (producing a 1 as output), or it could remain in $S0$ (producing a 0).

It is easy to see that the bit sequences produced by FSMs are *correlated*. Each bit cannot be set independently of the others. For example, the FSM of Fig. 7.10 (a) can never produce two consecutive 1s at the output. The FSM of Fig. 7.10 (b) produces a 0 every third bit.

Virtually any digitally implementable system, including complex communication protocols, encoders and decoders, error-correcting codes, *etc.*, can be represented as an FSM. This enables us to use FSMs as a general way to specify arbitrary coding schemes/bit correlations for eye diagram analysis.

7.3.2.3 Coding schemes: different eye openings for different bits

When a coding strategy is employed as part of a communication scheme, different received bits can have widely different eye diagrams.

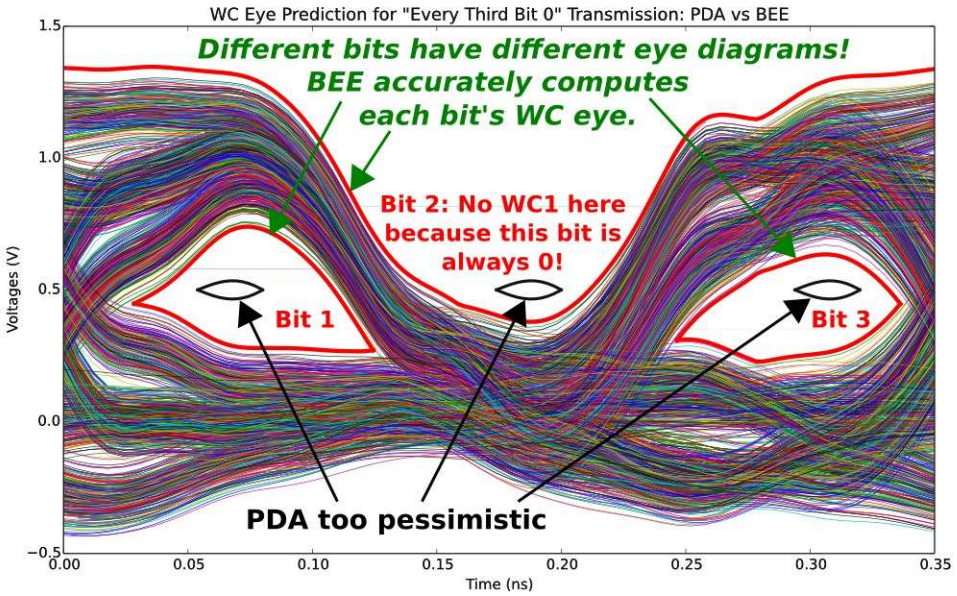


Figure 7.11: An example to show that a transmit-side coding scheme can have a profound impact on receive-side eye diagrams: different received bits can have widely different eye diagrams. In this example, the encoder transmits a 0 every third bit (Bit 2, Bit 5, Bit 8, *etc.*), which significantly improves the eye diagrams for all bits at the receiver. PDA does not recognize this and is too pessimistic, whereas BEE accurately accounts for each individual bit, producing worst case eye diagrams that are consistent with Monte-Carlo simulations. This is true even for the bits that are always 0, where the question of a worst case 1 does not arise.

For example, consider the coding scheme represented by the FSM of Fig. 7.10 (b). This scheme produces a 0 at every third bit. Suppose this FSM feeds into an LTI channel (*e.g.*, an RLGC chain). The eye openings for the received bits at the end of the channel will clearly be very different from one another. For example, every third bit (constrained to be a 0), will have only a one-sided eye opening (as there is no WC1 to consider). Similarly, the bit immediately preceding the constrained-0 bit will have a completely different eye opening from the bit immediately following the constrained-0 bit. Fig. 7.11 depicts the eye diagrams for these bits, along with the eye openings predicted by PDA (black) and BEE (red).

PDA only accounts for the channel, not the coding scheme. Therefore, it predicts the same pessimistic eye for all bits, as Fig. 7.11 depicts. BEE, on the other hand, fully accounts for the underlying coding strategy as well as the channel, and therefore accurately predicts the worst-case eye opening for each received bit (including the one-sided "eye" for the constrained-0 bit, as shown in Fig. 7.11).

7.3.2.4 BEE: Worst case eye prediction for correlated bits/coding schemes

Following Section 7.3.2.2, we model the underlying coding scheme as an FSM, which produces a correlated bitstream that feeds into an LTI channel (Fig. 7.9). We now describe BEE's algorithm for computing WC eye openings for such systems.

The problem statement is almost identical to the PDA optimization problems (7.4) and (7.5). The crucial difference, however, is that unlike PDA, the bits $\{b_k\}$ cannot be independently chosen as 0 or 1; the bits have to correspond to valid output sequences that can be produced by the underlying FSM.

As mentioned in Section 7.3.1, the core idea behind BEE is to label each state in the FSM with a realistic worst-case scenario, *i.e.*, a sequence of bits terminating at the given state with the highest cumulative cost. This is best illustrated by an example. Let us imagine that the FSM in question is the one shown in Fig. 7.10 (a). Further, let us assume that we are solving the WC0 optimization problem, and that we would like to maximize $5b_0 + 3b_1 + 2b_2$, where bit b_i is the output of the FSM at time i (assume that we begin at time 0 at state S0). If we were executing PDA, the solution would be simple; since all the multiplying coefficients are positive, we choose all the 3 bits (b_0, b_1, and b_2) to be 1. However, this solution is overly pessimistic, because we know from Section 7.3.2.2 that our FSM can never produce 2 consecutive 1s.

As Section 7.3.1 mentions, BEE solves the above problem via *dynamic programming*. In the above example, BEE starts with a simpler problem: what is the maximum value of $5b_0$ that leaves us in each state (S0, S1, and S2) at time 1? Clearly, the answer is 0 for state S0 (corresponding to the bit sequence 0), 5 for state S1 (corresponding to the sequence 1), and we cannot ever be in state $S2$ at time 1. At this stage, therefore, BEE tags the states S0, S1, and S2 with the optimal bit sequences 0, 1, and `UNDEFINED`, and the costs 0, 5, and `UNDEFINED` respectively.

Now BEE solves a slightly harder problem: what is the maximum value of $5b_0 + 3b_1$ terminating in each state at time 2? Clearly, to arrive at S0 at time 2, we must either reach

S0 at time 1 and stay there, or reach S1 at time 1 and then transfer to S0. The first option yields a maximum cumulative cost of 0, while the second yields a cost of 5 (a cost of 5 to reach S1 at time 1, known from the previously calculated optimal solution at S1, plus a cost of 0 going from S1 to S0). So BEE chooses the second option, and tags S0 with the sequence 10 and the cost 5. Note how BEE used the solution to the previous problem to build up a solution to the current one. This is the crux of BEE's dynamic programming algorithm: reusing solutions to previously solved simpler problems to gradually compute the entire WC eye. Thus, in this example, at the end of the second iteration, the states S0, S1, and S2 are tagged with the sequences 10, 01, and 10, and the costs 5, 3, and 5 respectively.

More generally, at each iteration, the bit sequence associated with each state is expanded by one bit (or set to `UNDEFINED`), and this bit is calculated from previously computed worst case costs. The iterations are continued until the required optimization problem is solved, which occurs when the lengths of the associated bit sequences exceed the memory of the underlying channel. At this point, the computed bit sequences are provably the worst-case scenarios for the link, and the corresponding worst-case eye openings are returned.

For instance, in the above example, at the end of the final iteration, the states S0, S1, and S2 are tagged with the sequences 100, 101, and 100, and the costs 5, 7, and 5 respectively. Picking out the maximum, the worst case cost is 7, corresponding to the worst case bit sequence 101. By contrast, if we had used PDA, the worst case cost returned would have been a pessimistic 10, corresponding to the (impossible) bit sequence 111.

Thus, not only is BEE's WC prediction *provably exact*, but it can also generate a certificate specifying the exact sequence of correlated bits corresponding to the WC outcome. This is a very useful property; in addition to providing an independently verifiable mathematical guarantee, it also serves to benchmark other commonly used heuristics/algorithms for WC eye estimation.

We note that our implementation of BEE includes mechanisms for handling asymmetric rise/fall times, jitter, stochasticity and parameter variability, *etc.*. Each of these is accounted for by adding extra functionality to the basic algorithm described above. Our results below illustrate these additional capabilities of BEE using real-world examples and test-cases.

We now apply the algorithms and techniques discussed above to perform worst case and stochastic analysis of eye diagrams in systems that arise in Signal Integrity, I/O and high-speed communication applications.

7.3.3 Example: Worst case eye analysis of a $(7, 4)$-Hamming encoded system

We now apply BEE to carry out WC eye diagram analysis of a $(7, 4)$-Hamming encoded communication scheme (a commonly used parity-based error correcting code), over an LTI channel composed of a chain of RLGC units (a lumped transmission line model) that captures inter-symbol interference, crosstalk, overshoot/undershoot, dispersion, *etc.*.

The $(7, 4)$-Hamming encoder accepts 4 data bits (d_1 – d_4) in parallel, and outputs a serial stream of 7 Hamming-encoded bits (p_1 through p_7). The relationship between the output bits $\vec{p}$ and the data bits $\vec{d}$ is given by the following:

$$\underbrace{\begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \end{pmatrix}}_{\vec{p}} = \underbrace{\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{G}} \underbrace{\begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{pmatrix}}_{\vec{d}} \quad \text{(modulo 2).} \tag{7.6}$$

We now construct an FSM for the $(7, 4)$-Hamming coding scheme above. To do so, we observe that not all 7-bit combinations are valid Hamming codewords. Indeed, although there are 128 possible combinations of 7 bits, only 16 of these represent valid codewords. These can be represented using a *prefix tree* structure, as shown in Fig. 7.12.

The root of the prefix tree denotes the start state of our FSM. Every proper prefix of each valid Hamming codeword corresponds to an internal state in the tree (or a state in the FSM). The 16 leaves of the tree (representing valid Hamming codewords) are looped back to the start state of the FSM, which resets the encoder every 7 bits, making it ready to produce the next 7-bit codeword. It can be verified that this FSM produces exactly the same bit sequences that a $(7, 4)$-Hamming encoder is capable of generating; so this is our model for the correlated bitstream entering the channel.

The channel is an analog LTI system that consists of a chain of 30 RLGC units, as shown in Fig. 7.13 below.

The pulse response of the above channel is shown in Fig. 7.14.

For the above "$(7, 4)$-Hamming encoder + channel" system, we now determine the WC eyes at the receiver using BEE. We also validate the worst case eyes produced by BEE – by carrying out a large number of Monte Carlo simulations of the system. Furthermore, we apply PDA to the above system (which pessimistically assumes that the bits are all uncorrelated), and compare its predictions against BEE's.

The results are shown in Fig. 7.15. Note that each of the 7 bits produced by the encoder (per codeword) gives rise to a differently shaped eye opening, as discussed in Section 7.3.2.3. This is a direct consequence of the correlated nature of the bits – which PDA is unable to account for. On the other hand, as seen from Fig. 7.15, BEE accurately reproduces the true shape of the WC eye for each received bit.

Further, in the experiment above, BEE was an order of magnitude faster than Monte-Carlo. Also, even though we simulated thousands of bits in our Monte Carlo runs, we were unable to generate the worst case. But once the worst case sequences were discovered by BEE, we were able to include them in our Monte Carlo runs and confirm that these sequences indeed led to the worst case outcomes predicted by BEE. This is true for many real-world

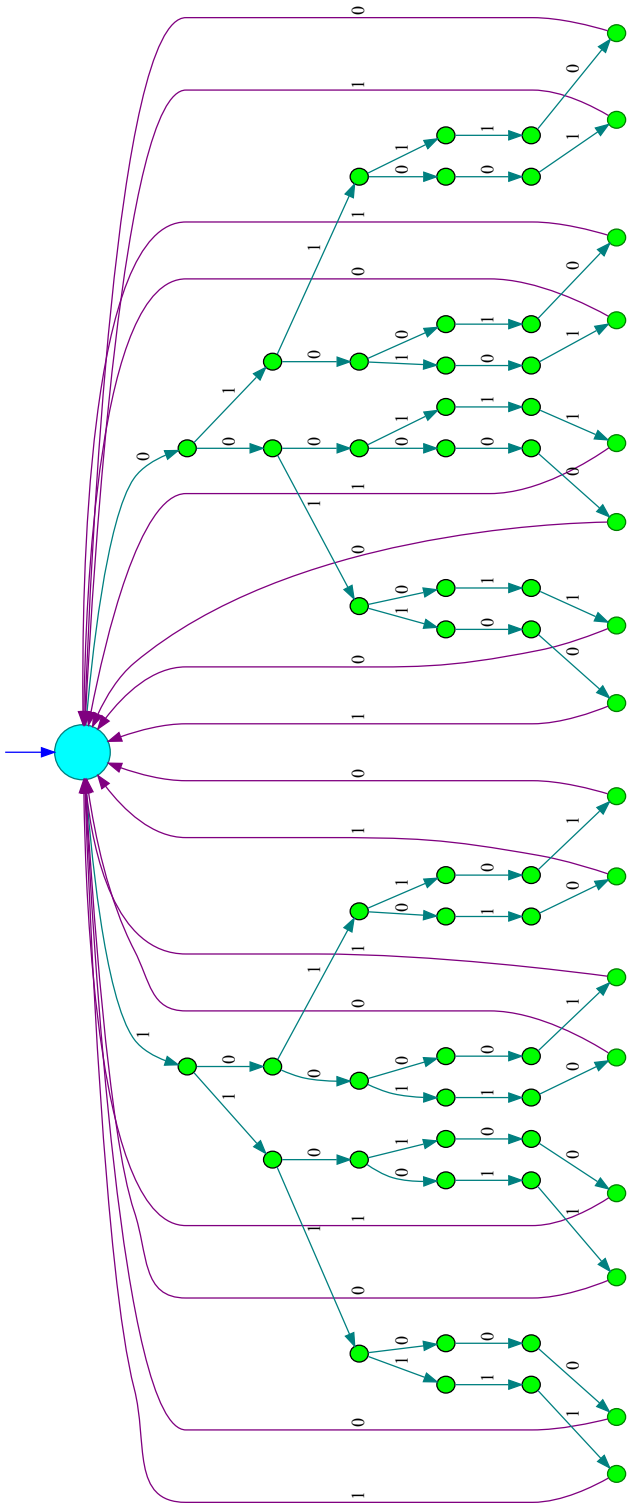


Figure 7.12: The $(7, 4)$-Hamming encoder FSM based on the 16 codeword prefix tree.

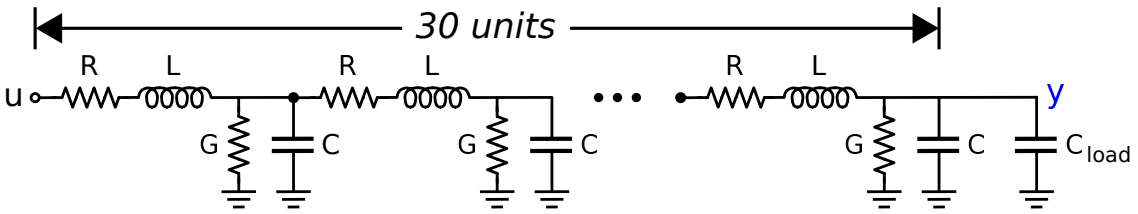


Figure 7.13: A 30-unit RLGC chain used to model the analog channel following the $(7, 4)$-Hamming encoder.

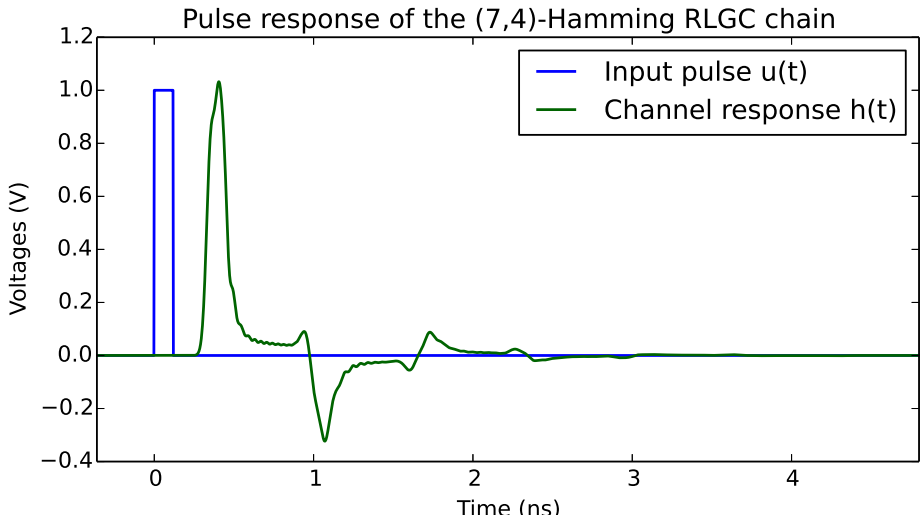


Figure 7.14: Pulse response of the 30 unit RLGC chain analog channel following the $(7, 4)$-Hamming encoder.

systems: the number of Monte Carlo runs needed to reliably generate worst case outcomes is often completely impractical from a computational standpoint. In such situations, BEE can offer valuable guidance in directing the search for the worst case.

7.3.4 Example: Worst case eye analysis of an $8b/10b$-SERDES system

We now apply BEE to an $8b/10b$ SERDES encoder, followed by a behavioral channel macromodel (a cascade of tanh(.) smoothened delays).

In this system, the encoder accepts as input 8 data bits in parallel, and produces as output 10 serialized bits. The encoded bits have some very desirable correlation properties that often result in significantly improved WC eye diagrams at the receiver. For example, it is guaranteed that no more than 5 consecutive 0s or 1s will occur at any time in the bitstream.

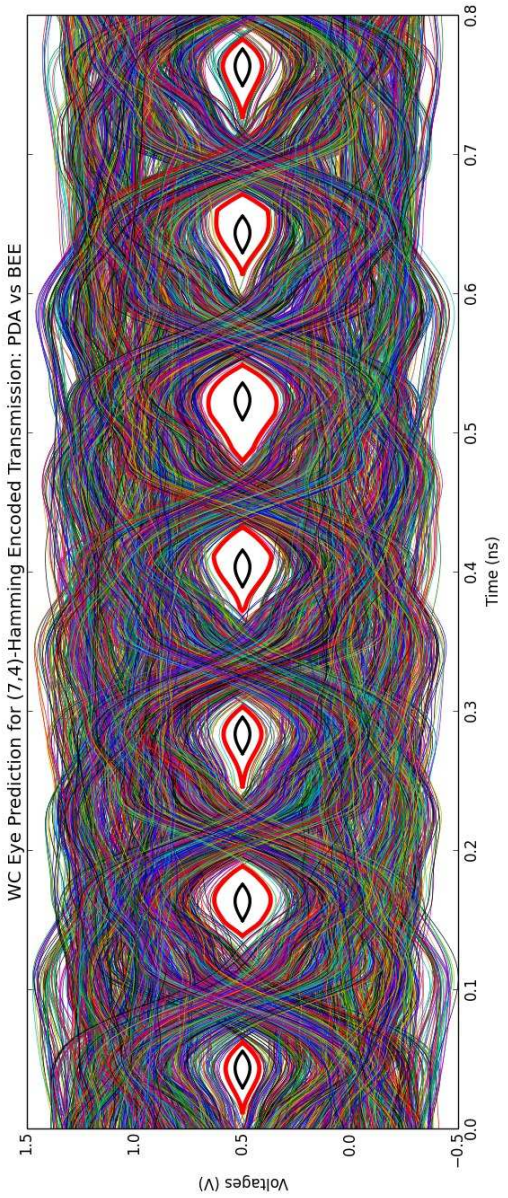


Figure 7.15: Eye diagrams predicted by Monte Carlo simulation (various colors), by PDA (black), and by BEE (red) for the $(7, 4)$-Hamming encoded communication scheme of Section 7.3.3. It is seen that BEE is able to produce exact worst case eye diagrams for the given system, unlike the overly pessimistic eye diagrams predicted by PDA.

This "normalization" of bits has a very positive impact on worst case eye diagrams, a fact that is completely ignored by PDA.

The $8b/10b$ encoding is carried out in two stages, (i) a $5b/6b$ stage that encodes the first five bits into a 6-bit word, and (ii) a $3b/4b$ stage that encodes the last three bits into a 4-bit word. To avoid a streak of more than 5 consecutive 0s or 1s, the encoder keeps track of a quantity called the *Running Disparity (RD)*, defined as the number of 1s minus the number of 0s in the bitstream produced thus far (for details, see [85]).

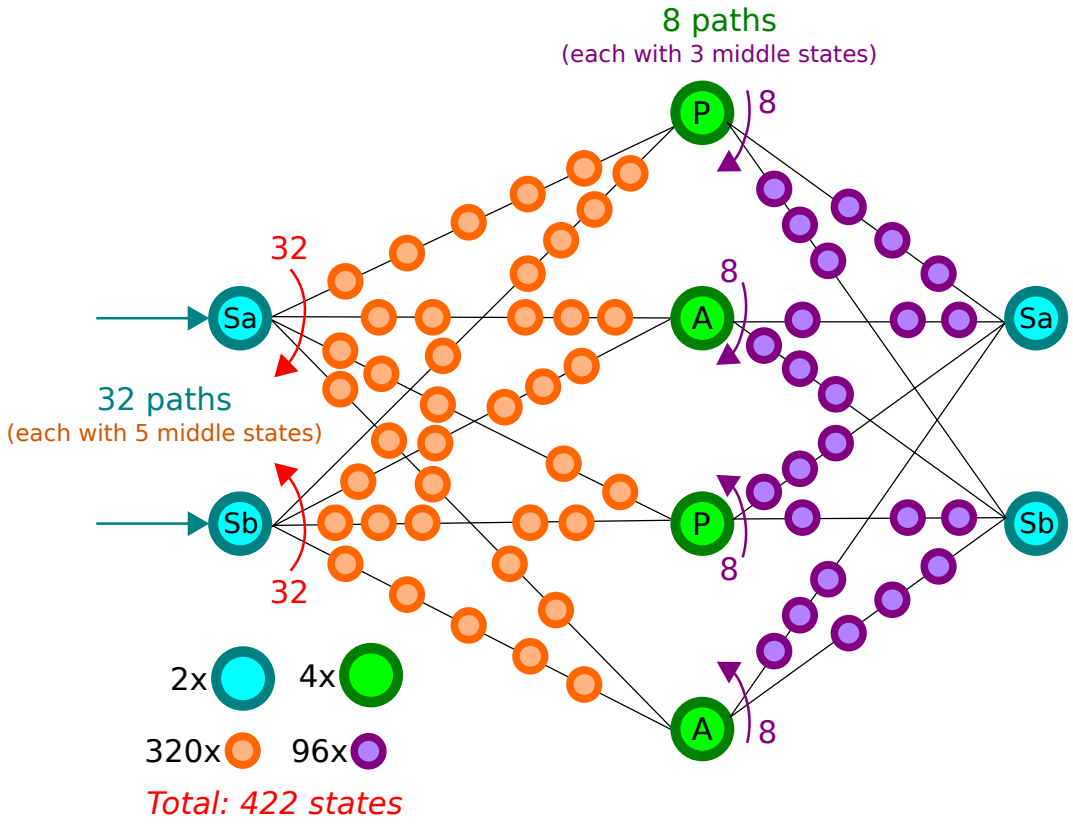


Figure 7.16: An $8b/10b$ encoder FSM based on $5b/6b$ and $3b/4b$ codes.

The FSM model for the $8b/10b$ SERDES encoder is shown in Fig. 7.16. The FSM has two start states (labelled Sa and Sb in the figure), corresponding to RD being -1 and $+1$ respectively. As seen from the figure, the $5b/6b$ encoding stage is implemented using 64 FSM paths (32 emanating from Sa and 32 from Sb), each with 5 intermediate FSM states and a terminal state that is one of the 4 green states shown in the figure. This is followed by the $3b/4b$ encoding stage, which consists of 32 additional FSM paths (8 paths emanating from each of the 4 green FSM states), that eventually loop back to Sa or Sb after traversing 3 intermediate states each.

The channel following the above FSM is a tanh(.) smoothed cascaded delay macromodel that is often used to capture the dynamics of transmission lines with multiple reflections,

dispersion, overshoot/undershoot *etc.*. The pulse response of this channel $h(t)$ is given by the following equation:

$$h(t) = \sum_{i=0}^{N-1} \alpha_i \left(\frac{1 + \tanh(k(t - t_i^{(1)}))}{2} \right) \left(\frac{1 - \tanh(k(t - t_i^{(2)}))}{2} \right). \tag{7.7}$$

Each term in the above summation represents a smooth "dead delay", with amplitude α_i active during the time interval $[t_i^{(1)}, t_i^{(2)}]$. The pulse response of the channel is shown in Fig. 7.17.

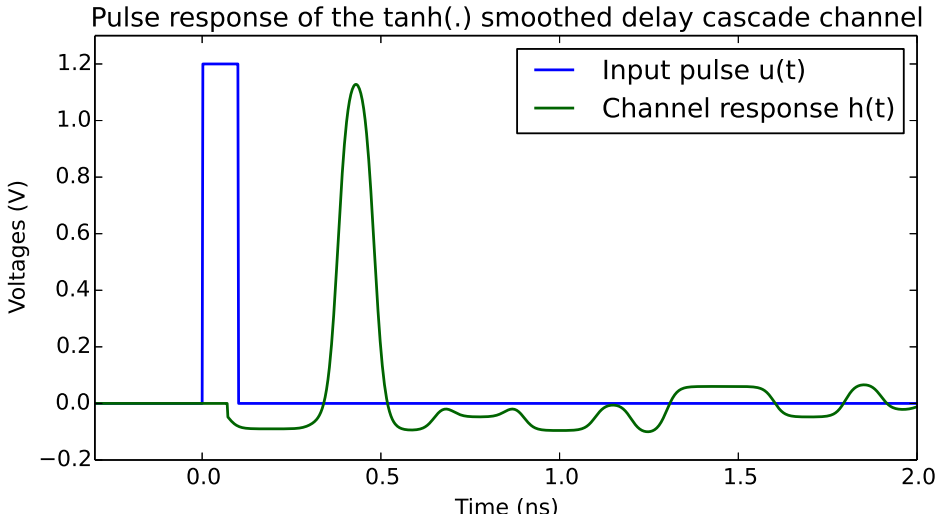


Figure 7.17: Pulse response of the tanh(.) smoothed cascaded delay channel following the $8b/10b$ SERDES encoder.

Similar to our analysis of the $(7, 4)$-Hamming encoder Section 7.3.3, we now use BEE to determine the WC eye diagram for the $8b/10b$ SERDES system above. And just like the previous case, we see from Fig. 7.18 that BEE is able to accurately reproduce the true shapes of the WC eye diagrams for the given system (for all the 10 bits), whereas the predictions made by PDA are overly pessimistic.

7.3.5 Example: Worst case eye diagrams for a Reed-Solomon encoded system

We now apply BEE to a communication system that uses a Reed-Solomon encoding of the transmitted bits.

The FSM part of the system is a straightforward Reed-Solomon encoder. The encoder has four parameters N, k, n, and s. The parameter N is a prime number, and all computations done by the encoder are carried out in $GF(N)$ (the standard Galois Field containing the

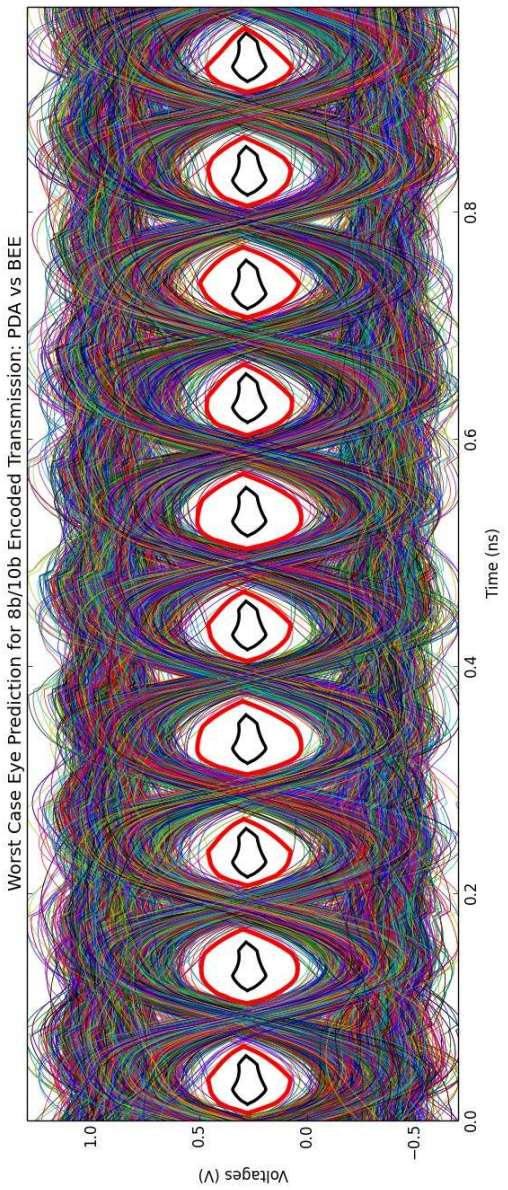


Figure 7.18: Eye diagrams predicted by Monte Carlo simulation (various colors), by PDA (black), and by BEE (red) for the $8b/10b$ SERDES encoded communication scheme of Section 7.3.4. It is clearly seen that the approach taken by BEE is able to produce exact and accurate WC eye diagrams for the given system, unlike the overly pessimistic eye diagrams predicted by PDA.

numbers 0 through $N-1$). The message to be transmitted consists of k symbols, where each of these symbols is an integer belonging to $GF(N)$, encoded as an s-bit number. Given the message consisting of k symbols m_0 through m_{k-1}, the Reed-Solomon encoder first finds the unique polynomial $P(x)$ of degree $k-1$ over the field $GF(N)$ such that $P(i) = m_i \; \forall 0 \leq i \leq k-1$. Having determined $P(x)$, the encoder then computes the symbols $P(0)$ through $P(n-1)$, where $n > k$, encodes these n symbols using s bits per symbol, and transmits these ns bits across the channel. Thus, only a small fraction of all combinations of the ns bits will be valid transmit symbols. In the example presented below, we use $(N, k, n, s) = (3, 2, 3, 2)$.

The channel following the encoder above is modelled as a 25-unit RLGC chain.

We designed an FSM to mimic the encoder above, and then applied BEE to carry out Boolean/LTI co-analysis of the "encoder + channel" system above, to predict worst case eye diagrams for this system. As with the previous two examples, we plotted the eye diagrams predicted by BEE, by PDA, and by Monte-Carlo simulation over one another.

The result is shown in Fig. 7.19. Again, we see that the eye diagrams predicted by BEE closely match those obtained from Monte-Carlo simulation, while the eye diagrams predicted by PDA are once again overly pessimistic. Furthermore, BEE was an order of magnitude faster than Monte-Carlo simulation for this system.

7.3.6 Example: Worst case eye diagrams for a Pre-emphasis/De-emphasis based communication scheme

Our fourth and final example involves a system implementing a pre-emphasis and de-emphasis scheme for its communication. The FSM part of this system is conceptually different because the symbols produced by the FSM are no longer just 0 or 1.

This is because, in a system that uses pre-emphasis and de-emphasis, the voltage that is transmitted across the channel is not always either 0 Volts or V_{DD} Volts. Instead, the transmitted voltage depends on not just the current message bit, but also potentially on previous message bits. For example, if the current bit is 1 and the previous bit is also 1, one may choose to de-emphasize the second bit, *i.e.*, transmit a lower voltage than V_{DD}. Similarly, if two successive message bits are both 0, then the second 0 may be de-emphasized a little (*i.e.*, a voltage that is slightly higher than 0 Volts may be transmitted). In cases where the current bit differs from the previous bit, the current bit may be emphasized a little (for example, a 1 (0) may result in a voltage that is slightly higher (lower) than V_{DD} (0) Volts). This is the key principle behind pre-emphasis and de-emphasis, and to model this accurately, the FSM needs to be able to output *multiple symbols* corresponding to the various possible emphasized and de-emphasized voltages. The Boolean/LTI co-analysis procedures underlying BEE also need to be modified to account for this.

Fig. 7.20 shows a multi-symbol Mealy machine FSM that we use to model the pre-emphasis/de-emphasis based encoder. The FSM has just 2 states, but it has 4 output symbols $\{A,\ B,\ C,\ D\}$, each corresponding to a different transmit voltage as shown in the figure. The FSM above is followed by a 25-unit RLGC chain.

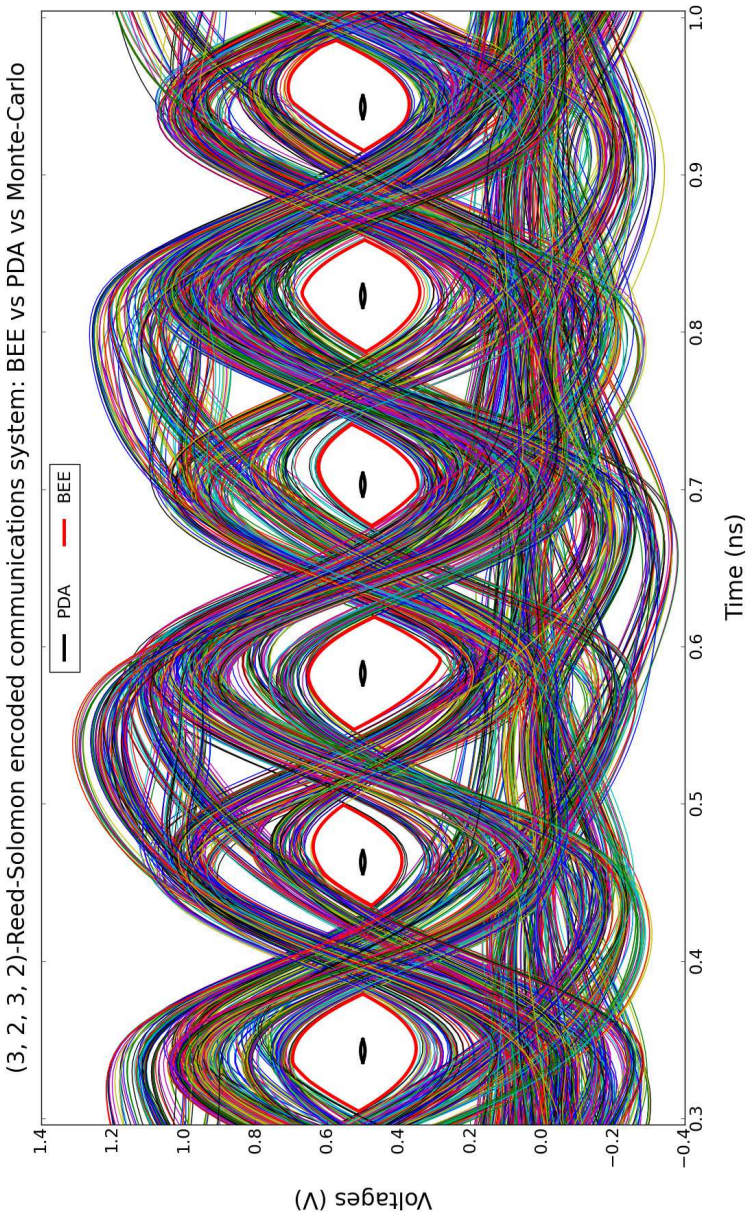


Figure 7.19: Eye diagrams predicted by Monte Carlo simulation (various colors), by PDA (black), and by BEE (red) for the Reed-Solomon encoded communication scheme of Section 7.3.5. It is once again clear that BEE is able to produce exact WC eye diagrams for the given system, unlike the overly pessimistic eye diagrams predicted by PDA.

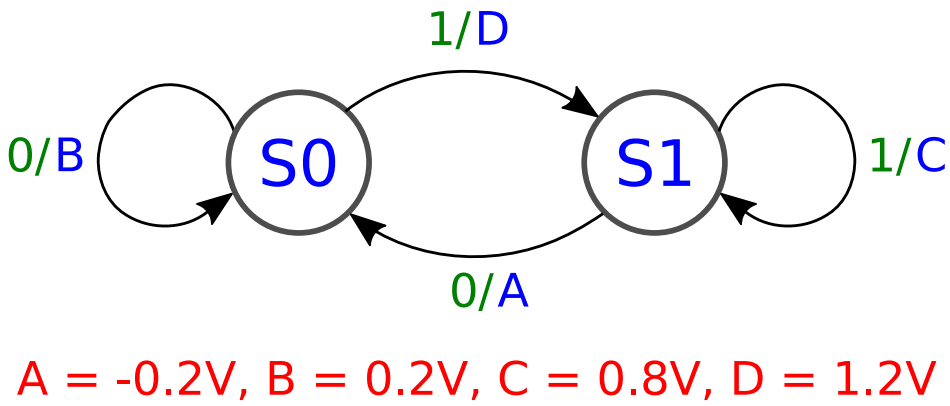


Figure 7.20: Multi-symbol Mealy machine FSM used to implement pre-emphasis and de-emphasis.

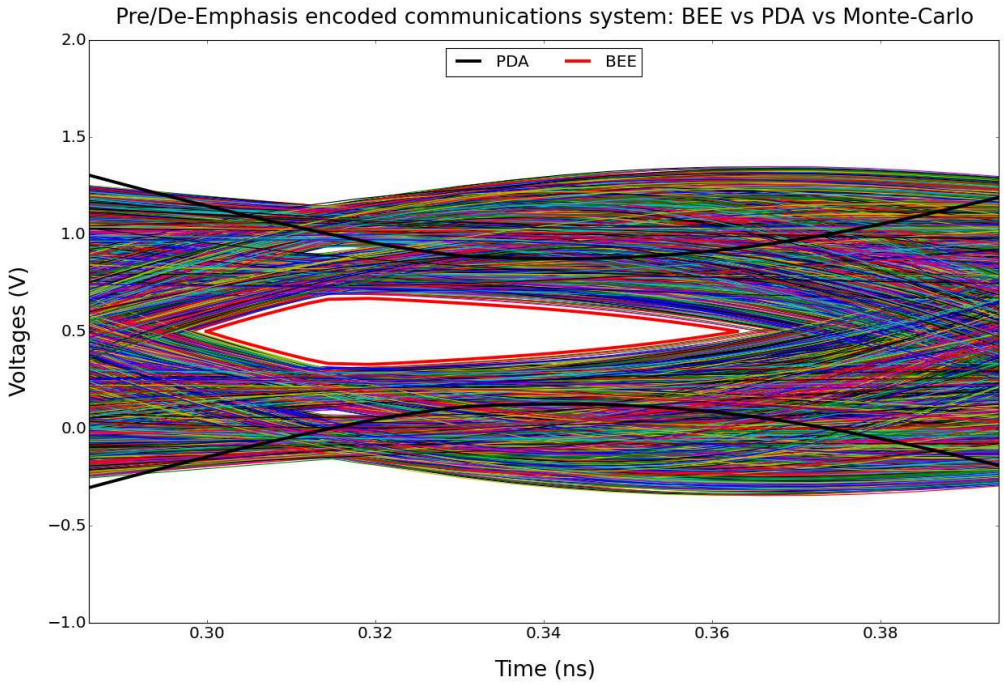


Figure 7.21: Eye diagrams predicted by Monte Carlo simulation (various colors), by PDA (black), and by BEE (red) for the pre-emphasis/de-emphasis based communication scheme of Section 7.3.6. In this case, BEE accurately predicts the eye opening and tallies well with Monte-Carlo simulations, whereas PDA fails claiming that there exists no eye opening.

When we used PDA and BEE to predict the worst case eye diagrams for the above system, we encountered an interesting situation: we found that as usual, BEE was able to predict the eye diagram exactly (tallying well with Monte-Carlo simulations), but PDA was so pessimistic that it failed to predict any eye at all – *i.e.*, PDA claimed that the system would have no eye opening. This is shown in Fig. 7.21.

7.3.7 Example: Jitter analysis and worst case eye diagrams for the $(7, 4)$-Hamming system

We now apply BEE to carry out jitter analysis of the systems above. Our model for jitter is a random time shift of the channel output. Thus, if we denote the output of the LTI channel by $y(t)$, then the jittered output is given by $y(t - J(t))$, where $J(t)$ is the (time-varying) jitter in the system.

If $J(t)$ is *bounded*, *i.e.*, $J(t)$ always lies in a range (say, between $J_{\min}$ and $J_{\max}$), then BEE's worst case dynamic programming algorithm can be modified to compute the worst case eye in the presence of jitter. Thus, by setting different bounds on the jitter, we can use BEE to compute the *jitter tolerance* of each bit in the system, *i.e.*, the gracefulness of degradation of the eye associated with each bit as jitter is gradually increased. This is shown in Fig. 7.22, where we apply BEE to predict the worst case eye of each bit in our $(7, 4)$-Hamming system, as jitter increases from 0 to $\pm 10\%$ of the bit period T in gradual steps. At jitter being 0, the worst case prediction simply reduces to the worst case prediction without jitter (the black contours in the figure). However, as jitter increases, the eye of each bit starts collapsing – into smaller and smaller concentric eyes as shown in the figure. We believe that this kind of detailed information regarding the tolerance of each bit to jitter can help designers come up with optimal pre-emphasis/de-emphasis strategies, placement of active buffers and boosters, more accurate BER analysis, *etc.*.

7.3.8 Example: Stochastic analysis of eye diagrams with parameter variability for the $(7, 4)$-Hamming system

We now introduce a source of randomness into our analysis, namely parameter variability.

To model parameter variability, we sampled the parameters R, L, G, and C of the $(7, 4)$-Hamming system from independent Gaussian distributions with suitable mean and variance. For a large number of such samples, we computed the pulse responses $h(.)$ as functions of time, and plotted them on top of one another, yielding Fig. 7.23. From such data, we generated a statistical model for $h(.)$ and then applied BEE to carry out combined stochastic analysis of this model with a probabilistic FSM model.

The resulting statistical eye characterization is shown in Fig. 7.24 (please see the figure caption for more details). Thus, at a glance, the visualizations produced by BEE enable one to quickly grasp important features of the statistics of the underlying system in the presence

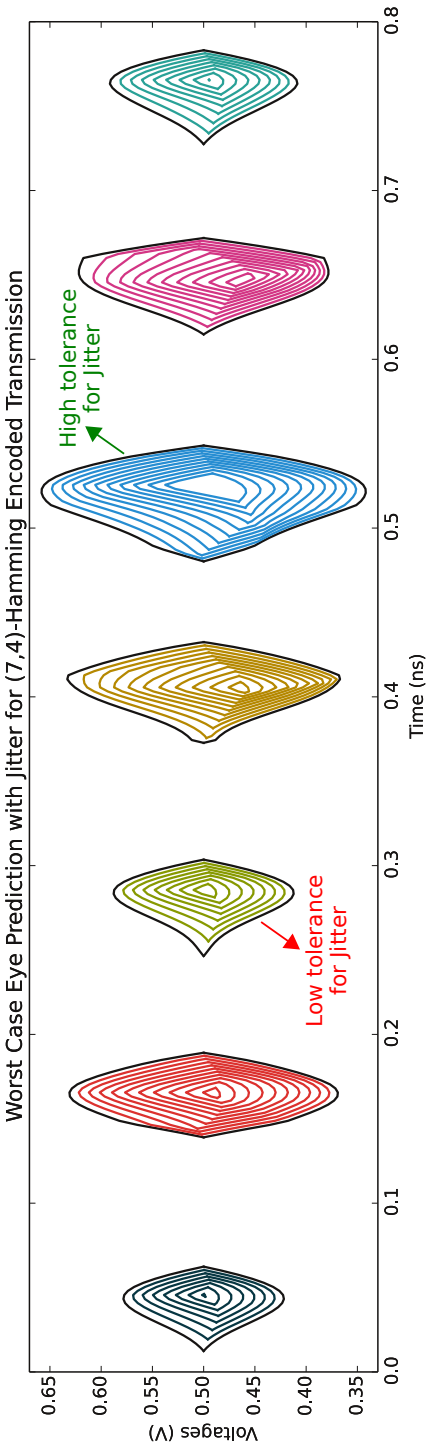


Figure 7.22: Using BEE to predict the jitter tolerance/jitter margin of each bit in the $(7, 4)$-Hamming system. The outermost eye diagram for each bit is the worst case eye in the absence of jitter. As jitter is gradually increased (from 0 to 10% of the period T), the eyes start shrinking until the opening completely vanishes.

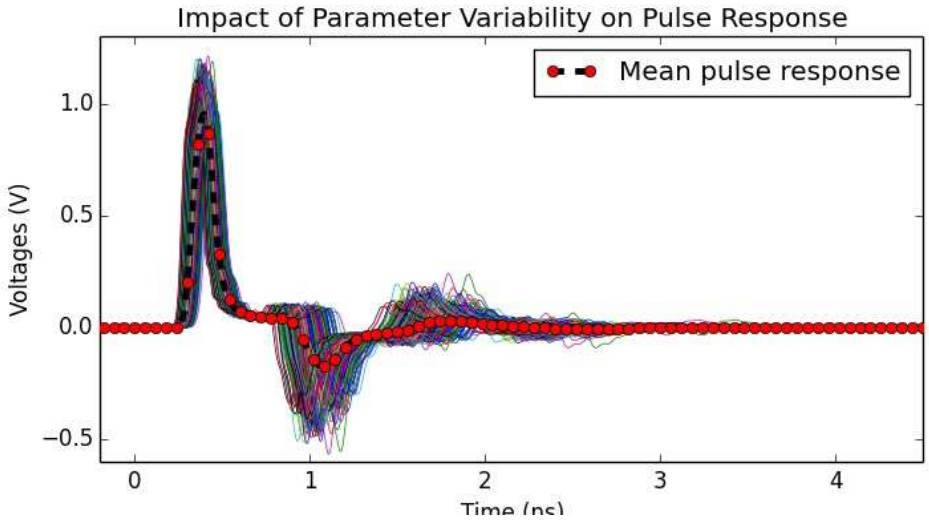


Figure 7.23: Time-varying distribution of pulse response as a result of parameter variability.

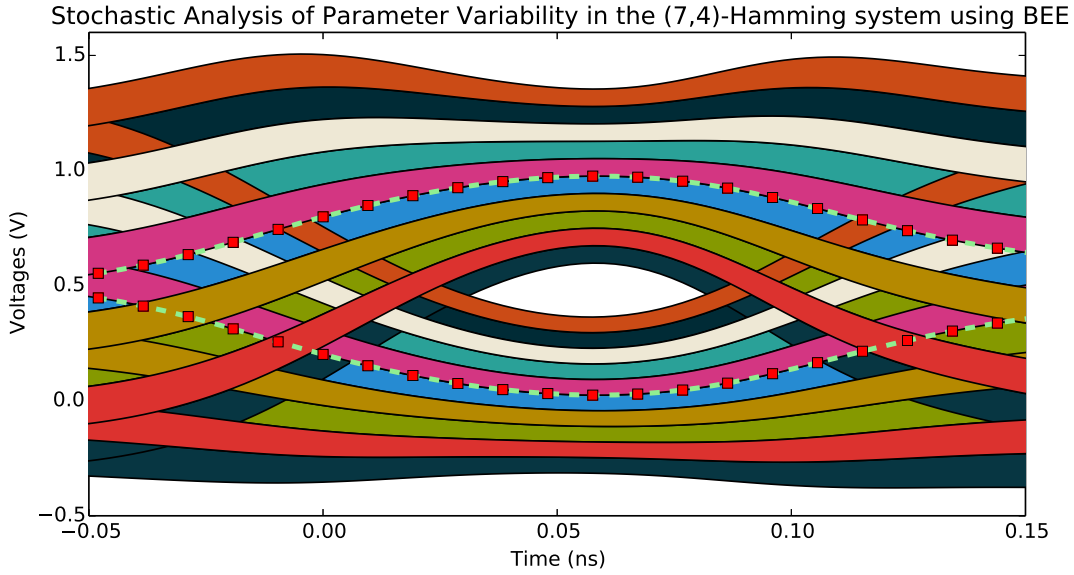


Figure 7.24: Statistical characterization of a $(7, 4)$-Hamming eye opening in the presence of parameter variability in R, L, G, and C of the underlying RLGC channel. The dashed line with the red markers is the time-varying mean of the channel output. The colored bands each have a thickness of 0.3σ, where σ is the time-varying standard deviation (square root of the variance) of the channel output.

of parameter variability. We believe that this analysis nicely complements the worst case analyses above.

7.4 Summary

In summary, we have presented 3 different use-cases for the models produced by Booleanization engines such as ABCD, in the context of AMS design.

The first and second use cases involve the well-known problems of high-speed AMS simulation and formal AMS verification respectively. Our discussion in this chapter points out that the accurate Boolean models produced by ABCD, by virtue of being fast and easy to simulate, and by virtue of being compatible with powerful state-of-the-art Boolean analysis, verification, and model checking engines such as ABC, have great potential to make significant inroads into both these areas of AMS design.

The third use case for Boolean models is in the development of custom analysis procedures for solving specific targeted problems that stand in the way of bug-free AMS design. One such example is the problem of estimating eye diagrams accurately in communication systems that employ coding strategies. We demonstrated above that this problem in fact permits a very elegant solution that combines elements of Boolean FSM analysis with elements of the theory of LTI systems (*i.e.*, Boolean/LTI co-analysis). Our solution to this problem, dubbed the Berkeley Eye Estimator or BEE, was demonstrably able to predict eye diagrams exactly and efficiently for several real-world communication sub-systems and encoding schemes.

Chapter 8

Conclusions and Future work

8.1 Conclusions

In conclusion, we would like to state the following:

- **The need for accurate modelling of AMS designs.** Accurate modelling, analysis, simulation, and verification of AMS designs are key problems faced by the EDA industry today. Previous approaches to the problem based on hybrid system methods and frameworks have not proved equal to the task, largely because of *poor modelling accuracy* and *poor scalability*. Hybrid system approaches boast the ability to represent and reason about both analog and digital (*i.e.*, both continuous and discrete) signals. However, existing hybrid system verification flows and methodologies scale very poorly with respect to the number of continuous variables involved, sometimes failing for systems featuring just a handful of continuous signals. Because continuous variables tend to be very expensive in a hybrid system, one is forced to *conserve* continuous variables by adopting overly simplistic models and unreasonable approximations for AMS components, with the effect that key performance limiting and error inducing real-world analog effects exhibited by the underlying design are not properly taken into account. This in turn limits the confidence an AMS designer can have in the guarantees issued by AMS verification tools. Therefore, in order to scale up AMS verification and enable effective bug-free AMS design, a new paradigm is needed that is able to accurately model the behaviour of AMS systems and components without incurring the huge penalties associated with having to reason about continuous quantities.

- **Booleanization is a promising solution.** We believe that Booleanization, *i.e.*, the practice of approximating continuous and mixed-signal dynamical systems using purely Boolean models such as truth tables and FSMs, offers a promising solution to the above problem. Because Booleanization produces models that only contain discrete quantities and purely Boolean operations, it eliminates the need to use expensive continuous variables. Moreover, as we have shown, it is possible to model the analog dynamics of

AMS designs to a high degree of accuracy just by increasing the fineness of discretization of the generated Boolean model. Furthermore, Booleanization opens the door to using powerful existing state-of-the-art formal verification and model checking tools (such as ABC) to formally verify AMS designs as well. To this end, we have developed ABCD, a suite of automated Booleanization tools, techniques, and algorithms capable of Booleanizing a wide variety of AMS designs.

- **DAE2FSM is effective at Booleanizing "digitalish" designs.** We developed DAE2FSM with a view to Booleanizing "digitalish" designs, *i.e.*, designs whose intended functionality is purely digital, but which nevertheless suffer from significant analog non-idealities that impact functional correctness as well as performance. DAE2FSM is based on an adaptation of Angluin's algorithm from computational learning theory, and our experiments on a variety of latches, flip-flops, and larger circuits such as counters indicate that DAE2FSM is quite effective at Booleanizing such "digitalish" designs, producing FSM abstractions that not only capture their ideal intended functionality, but also analog non-idealities exhibited by their real-world transistor-level implementations. However, DAE2FSM does not work well for genuinely analog systems, especially those featuring long "memories", which inspired us to develop non-black-box Booleanization techniques like ABCD-L and ABCD-NL.

- **ABCD-L is effective at Booleanizing LTI systems.** Our experiments in Chapter 5 indicate that ABCD-L is a powerful technique for Booleanizing a wide variety of LTI systems such as equalizers, filters, communication channels, power grid networks, *etc.*. Indeed, the Boolean models produced by ABCD-L not only capture the analog dynamics exhibited by these systems with high accuracy, but also achieve a good balance between accuracy and scalability by taking full advantage of the theory of LTI systems and eigen-analysis. Furthermore, ABCD-L is also well-suited for Booleanizing large LTI systems by applying it in conjunction with LTI model order reduction techniques.

- **ABCD-NL is effective at Booleanizing a large class of genuinely analog, genuinely non-linear AMS systems.** We developed ABCD-NL to be a powerful, general-purpose Booleanization tool that worked in situations where DAE2FSM and ABCD-L failed. By recognizing that a large class of non-linear AMS systems and components follow a simple DC/DC rule (*i.e.*, they respond to DC inputs with DC outputs), we were able to exploit this to develop an algorithm for Booleanizing such systems based on separating their DC dynamics from their transient dynamics. Our results obtained by applying ABCD-NL to a variety of circuits (including, for example, charge pumps, delay lines, ADCs and DACs, comparators, signalling systems, *etc.*) give us confidence that ABCD-NL is indeed a powerful Booleanization technique that produces accurate Boolean models even for non-linear AMS circuits without unduly sacrificing scalability.

- **Boolean models have a variety of interesting uses in AMS design.** We can identify three potential use cases for Boolean models (such as the ones produced by the Booleanization techniques above) in the context of AMS design. The first is high-speed AMS simulation. Because Boolean models are fast and easy to simulate (unlike continuous SPICE-level differential-algebraic equation based models), and because they can often capture analog dynamics to a high level of accuracy, we believe that in many situations, Boolean models are potentially usable as much faster, almost-as-accurate, drop-in replacements for SPICE models. The second use case is formal verification. We believe that Boolean models have the potential to be used in conjunction with existing tools, techniques, and methodologies for both Boolean analysis as well as hybrid system analysis to enable scalable formal verification of AMS designs. The final use case is the development of novel "custom" algorithms and analysis procedures for solving targeted problems in the AMS design space. An example of such a custom analysis technique is BEE, a Boolean/LTI co-analysis technique that we developed to accurately predict worst case and stochastic eye diagrams in modern high-speed communication sub-systems that employ coding strategies. In such systems, BEE is able to predict eye diagrams exactly (with neither undue optimism nor excessive pessimism), at a fraction of the computational cost of Monte-Carlo simulation.

8.2 Future work

We have also identified several directions for future work/research:

- **Direct construction of efficient circuit representations.** It is well-known that Boolean-circuit representations such as BDDs, AIGs, *etc.*, are typically much more efficient data structures for Boolean models compared to State Transition Graphs (STGs). And ABCD-NL does have support for some of these efficient formats and data structures. However, at the moment, ABCD-NL first constructs STGs, which can then be transformed into these more efficient data structures. This explicit state enumeration may not be feasible for large Boolean models. So, we would like to sidestep the STG generation and directly construct these more efficient circuit-based representations instead. If this is possible (and we believe it is), it promises to make ABCD significantly more efficient than it is right now.

- **Larger suite of Booleanization examples.** We would like to Booleanize a much larger set of circuits and systems than we have done so far. The key challenge we face in this regard is that, being in an academic institution, we have very limited access to industrial AMS circuits/systems with more than a few dozen transistors. Indeed, we had to design many of the circuits that we Booleanized in this dissertation ourselves, because we could not obtain real-world industry scale designs for academic research.

- **Booleanization of marginally stable (and in particular, oscillatory) systems.** At present, ABCD-NL only works for robustly stable systems that obey the DC/DC rule: given a DC input, the system should eventually respond with a DC output. Marginally stable systems are an important and practically relevant class of AMS systems that do not obey this rule. Such systems play an important rôle in PLLs and other clock and data recovery circuits. Therefore, we would like to either extend the techniques underlying ABCD-NL, or develop new ones, for Booleanizing such systems as well. In particular, oscillators are an important subclass of marginally stable systems. At present, we do not have a technique for Booleanizing oscillators. However, we believe that this Booleanization is indeed possible (and perhaps not very difficult), and we would like to develop techniques for it in the future.

- **Template-based Booleanization.** We would like to derive efficient Booleanization "templates" for a variety of commonly occuring AMS circuits and components. For example, we would like to develop a template for OpAmps, another template for ADCs, another for SERDES systems, *etc.*. This way, we would be able to enable quick automated Booleanization of virtually any AMS design just by "fitting" a Boolean model to a set of parameters. This "cookie cutter approach" may also have significant implications for verification.

- **Property-directed Booleanization.** We would like to develop a system where, instead of trying to capture analog wave shapes in glorious detail, we Booleanize only the relevant details based on the property we are trying to verify using the generated Boolean model.

- **Booleanizing metastability.** Metastability is another important effect, exhibited by systems featuring unstable DC operating points, that we have not quite been able to Booleanize effectively thus far. We would like to develop techniques for doing so in the future.

- **The loading problem.** This has to do with composing Booleanized AMS systems together. It is well-known that when one AMS system's output is fed as input to another AMS system, the combined system can behave completely differently from what a "unidirectional signal flow graph" (USFG) model (which assumes no loading) would suggest.

  For example, let us consider two systems. The first is an RC chain (shown at the top left of Fig. 5.4), whose output is buffered and used as the input to another identical RC chain. This is a USFG-type system, because the output of one circuit is buffered before it is fed to another. The second system is an RC chain containing two RC units (as depicted in Fig. 5.6); this is the non-buffered version of the first system.

  The point behind the loading problem is that the two systems above behave very differently. They have different DAEs, different eigenvalues, different ISI/dispersion characteristics, *etc.*. Therefore, they will respond to similar inputs very differently.

The reason why the two systems are so different is that, in the first system, the entire current that flows through the resistor in the first RC unit also flows through the capacitor in this unit (from Kirchoff's current law). In other words, the second unit does not draw any current from the first; it merely *observes* the voltage at the first unit's output, without *affecting* this voltage in any way. But in the second system, only part of the current that flows through the resistor in the first RC unit also flows through the capacitor of this unit; the rest of the current now flows into the *second* unit. As a result, just by virtue of wiring the input of the second unit to the output of the first, the second unit begins to draw current away from the first. Therefore, the voltages and currents that are internal to the first unit undergo a profound change. In analog design parlance, we say that the input impedance of the second unit *loads* the first unit, and hence changes its voltages and currents.

Therefore, if we wanted to Booleanize the RC chain above, we cannot simply Booleanize the two individual RC units and then compose the two Boolean models together; even if we had Boolean models for the individual RC units, we would have to Booleanize the RC chain from scratch, as a completely different system in its own right.

If possible, we would like to avoid this. We would like to follow a modular approach where we Booleanize small individual AMS components, and then somehow compose these Boolean models together, to accurately reflect the behaviour of larger AMS components and systems. At present, unless the interfaces between such components are buffered, we would lose accuracy if we did this. For non-buffered interfaces (which are the norm in most AMS circuits), we would need to rethink the way we build our Boolean models and compose them. We believe that the research problem here is to make a precise connection between composing together SPICE-level AMS systems (*i.e.*, DAEs), and composing together their Booleanized versions.

- **Timing analysis using ABCD.** We would like to apply ABCD-generated models to carry out timing analysis of AMS systems, taking advantage of the fact that timing analysis is a very special case of the more general AMS verification problem, which may well be solvable more easily in practice.

- **Explore formal AMS verification more deeply.** We would like to carry out significant additional work on formally verifying AMS designs using Booleanized models. We believe that at the moment, we are still in the very initial stages of this field of research, somewhat akin to the state of Boolean model checking and verification before the advent of symbolic model checking algorithms. Therefore, over the next few years, we would like to invest some time and research effort into developing specialized techniques, algorithms, and approaches that are specifically geared towards verifying analog and AMS systems (just as today's Boolean verification tools have a host of built in methods and heuristics, such as BMC, PDR, induction, structural hashing, *etc.*, specifically geared towards verifying typical real-world digital designs).

- **Word-level verification vs bit-blasting.** In the context of formally verifying AMS designs, we would like to explore the idea of word-level verification in more detail. At the moment, our verification methodology is based entirely on bit-blasting, which often leads to loss of important structural information about the models being verified. We believe that it may be possible to develop word-level theories (akin to satisfiability modulo theories) and solvers that take full advantage of domain specific features of the AMS verification problem.

- **ABCD-based debugging of AMS designs.** We would like to explore the question of how to fix an AMS design once a verification tool has discovered a bug in the Booleanized version of the design. We believe that this is a very intriguing problem that has deep and fundamental connections to the concepts of observability and controllability between the original AMS design and its Booleanized version. We believe that AMS test-generation is a closely related problem as well, and we would like to devote some time to studying and working on these kinds of problems.

- **Better coding, release practices.** We would like to significantly overhaul and improve the ABCD code base, since we are not particularly happy with its current organization (or lack thereof). For starters, we would like to make the code cleaner and more modular, and we would also like to reduce boilerplate code (we believe this would also make the code more efficient). Also, we would like to improve the documentation within the code. We would also like to improve the ABCD website, and make it faster, better organized, and easier to maintain and push updates to. We would also like to set up a regular open source release schedule. Finally, we would like to piece together bits and pieces of code from various places and integrate it all into a single open-source release repository.

Bibliography

[1] C. Gu and J. Roychowdhury. FSM model abstraction for analog/mixed-signal circuits by learning from I/O trajectories. In *A*SP-DAC '11: Proceedings of the 16^{th} Asia and South Pacific Design Automation Conference, pages 7–12, 2011.

[2] C. Gu. *Model order reduction of non-linear dynamical systems*. PhD thesis, The University of California, Berkeley, 2011.

[3] R. Parker. Analog design challenges in the new era of process scaling. At the 2012 International Workshop on Design Automation for AMS Circuits (co-located with ICCAD).

[4] Mixed-signal design challenges and requirements (a marketing paper published by Cadence). Downloadable at `https://www.cadence.com/rl/Resources/white_papers/mixed_signal_challenges_wp.pdf`.

[5] G. Taylor. Future of analog design and upcoming challenges in nanometer CMOS. Keynote address at the 2010 International Conference on VLSI Design.

[6] R. K. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *C*AV '10: Proceedings of the 22^{nd} International Conference on Computer Aided Verification, pages 24–40, 2010.

[7] J. Roychowdhury. *Numerical simulation and modelling of electronic and biochemical systems*. Now Publishers Inc., 2009.

[8] X. Li, P. Gopalakrishnan, Y. Xu, and L. T. Pileggi. Robust analog/RF circuit design with projection-based performance modeling. *I*EEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26(1):2–15, 2007.

[9] M. Krasnicki, R. Phelps, R. A. Rutenbar, and L. R. Carley. MAELSTROM: Efficient simulation-based synthesis for custom analog cells. In *D*AC '99: Proceedings of the 36^{th} annual ACM/IEEE Design Automation Conference, pages 945–950, 1999.

[10] B. Sterin, N. Een, A. Mishchenko, and R. Brayton. The benefit of concurrency in model checking. In *I*WLS '11: Proceedings of the 20^{th} International Workshop on Logic Synthesis, pages 176–182, 2011.

[11] G. Cabodi, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, D. Vendraminetto, A. Biere, K. Heljanko, and J. Baumgartner. Hardware model checking competition 2014: an analysis and comparison of solvers and benchmarks. *J*ournal on Satisfiability, Boolean Modeling, and Computation, 9:135–172, 2016.

[12] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 2ed*. Addison-Wesley, 2001.

[13] J. L Stensby. *Phase locked loops: Theory and applications*. CRC Press, 1997.

[14] M. Althoff, A. Rajhans, B. H. Krogh, S. Yaldiz, X. Li, and L. Pileggi. Formal verification of Phase Locked Loops using reachability analysis and continuization. In *I*CCAD '10: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pages 659–666, 2010.

[15] A. Ghosh and R. Vemuri. Formal verification of synthesized analog designs. In *I*CCD '99: Proceedings of the IEEE International Conference on Computer Design, pages 40–45, 1999.

[16] K. Hanna. Reasoning about real circuits. In *P*roceedings of the 7^{th} International Workshop on Higher Order Logic Theorem Proving and Its Applications, pages 235–253, 1994.

[17] K. Hanna. Reasoning about analog-level implementations of digital systems. *F*ormal Methods in System Design, 16(2):127–158, 2000.

[18] R. P. Kurshan and K. L. McMillan. Analysis of digital circuits through symbolic reduction. *I*EEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 10(11):1356–1371, 1991.

[19] W. Hartong, L. Hedrich, and E. Barke. Model checking algorithms for analog verification. In *D*AC '02: Proceedings of the 39^{th} annual ACM Design Automation Conference, pages 542–547, 2002.

[20] S. Steinhorst and L. Hedrich. Model checking of analog systems using an analog specification language. In *D*ATE '08: Proceedings of the ACM Conference on Design, Automation and Test in Europe, pages 324–329, 2008.

[21] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. *H*ybrid Systems, pages 209–229, 1993.

[22] A. Nerode and W. Kohn. Models for hybrid systems: Automata, topologies, controllability, observability. *H*ybrid Systems, pages 317–356, 1993.

[23] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *T*heoretical Computer Science, 138(1):3–34, 1995.

[24] T. A. Henzinger. The theory of hybrid automata. In *L*ICS '96: Proceedings of the 11^{th} Annual IEEE Symposium on Logic in Computer Science, pages 278–292, 1996.

[25] G. Al-Sammane, M. H. Zaki, and S. Tahar. A symbolic methodology for the verification of AMS designs. In *D*ATE '07: Proceedings of the ACM Conference on Design, Automation and Test in Europe, pages 249–254, 2007.

[26] M. H. Zaki, G. Al-Sammane, S. Tahar, and G. Bois. Combining symbolic simulation and interval arithmetic for the verification of AMS designs. In *F*MCAD '07: Formal Methods in Computer Aided Design, pages 207–215, 2007.

[27] E. Asarin, T. Dang, and O. Maler. d/dt: A verification tool for hybrid systems. In *C*DC '01: Proceedings of the 40^{th} IEEE Conference on Decision and Control, pages 2893–2898, 2001.

[28] S. Ray, J. Bhadra, T. Portlock, and R. Syzdek. Modeling and verification of industrial flash memories. In *I*SQED '10: Proceedings of the 11^{th} International Symposium on Quality Electronic Design, pages 705–712, 2010.

[29] T. A. Henzinger, P. H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *I*nternational Journal on Software Tools for Technology Transfer, 1(1):110–122, 1997.

[30] G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. *I*nternational Journal on Software Tools for Technology Transfer, 10(3):263–279, 2008.

[31] A. Chutinan and B. H. Krogh. Computational techniques for hybrid system verification. *I*EEE Transactions on Automatic Control, 48(1):64–75, 2003.

[32] S. Little, N. Seegmiller, D. Walter, C. Myers, and T. Yoneda. Verification of ams circuits using labeled hybrid petri nets. In *I*CCAD '06: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pages 275–282, 2006.

[33] S. Little, D. Walter, K. Jones, and C. Myers. AMS circuit verification using models generated from simulation traces. *A*utomated Technology for Verification and Analysis, pages 114–128, 2007.

[34] S. Little. *Efficient Modeling and Verification of Analog/Mixed-Signal Circuits Using Labeled Hybrid Petri Nets*. PhD thesis, University of Utah, 2008.

[35] S. Batchu. Automatic Extraction of Behavioral Models from Simulations of Analog/Mixed-Signal (AMS) Circuits. Master's thesis, University of Utah, 2010.

[36] M. Althoff, O. Stursberg, and M. Buss. Computing reachable sets of hybrid systems using a combination of zonotopes and polytopes. *N*onlinear Analysis: Hybrid Systems, 4(2):233–249, 2010.

[37] F. Wang. Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures. *I*EEE Transactions on Software Engineering, 31(1):38–51, 2005.

[38] C. Le Guernic and A. Girard. Reachability analysis of hybrid systems using support functions. In *C*AV '09: Proceedings of the International Conference on Computer Aided Verification, pages 540–554, 2009.

[39] R. E Bryant. Graph-based algorithms for Boolean function manipulation. *I*EEE Transactions on Computers, 100(8):677–691, 1986.

[40] W. H. A. Schilders, H. A. van der Vorst, and J. Rommes. *Model Order Reduction: Theory, research aspects and applications*, volume 13 of *M*athematics in Industry. Springer Verlag, 2008.

[41] C. W. Gear. *Numerical initial value problems in ordinary differential equations*. Prentice Hall, Inc., 1971.

[42] R. Alur, T. A. Henzinger, G. Laferriere, and G. Pappas. Discrete abstractions of hybrid systems. *P*roceedings of the IEEE, 88(7):971–984, 2000.

[43] D. Angluin. Learning regular sets from queries and counter-examples. *I*nformation and Computation, 75:87–106, November 1987.

[44] `http://www-device.eecs.berkeley.edu/bsim/`.

[45] K. V. Aadithya and J. Roychowdhury. DAE2FSM: Automatic generation of accurate discrete-time logical abstractions for continuous-time circuit dynamics. In *D*AC '12: Proceedings of the 49^{th} Design Automation Conference, pages 311–316, 2012.

[46] E. M. Clarke, T. A. Henzinger, and H. Veith, editors. *Handbook of model checking*. Springer-Verlag, 2011.

[47] C. L. Guernic and A. Girard. Reachability analysis of hybrid systems using support functions. In *C*AV '09: Proceedings of the 21^{st} International Conference on Computer Aided Verification, pages 540–554, 2009.

[48] A. Girard, C. L. Guernic, and O. Maler. Efficient computation of reachable sets of linear time-invariant systems with inputs. In *H*SCC '06: Proceedings of the 9^{th} International Conference on Hybrid Systems, Computation, and Control, pages 257–271, 2006.

[49] C. Tomlin, I. Mitchell, A. M. Bayen, and M. Oishi. Computational techniques for the verification of hybrid systems. *P*roceedings of the IEEE, 91(7):986–1001, 2003.

[50] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[51] `http://ptm.asu.edu/modelcard/HP/22nm_HP.pm`.

[52] `http://www.spiceopus.si/`.

[53] A. V. Karthik and J. Roychowdhury. ABCD-L: approximating continuous linear systems using Boolean models. In *D*AC '13: Proceedings of the 50^{th} Design Automation Conference, pages 63:1–63:9, 2013.

[54] B. Gustavsen and A. Semlyen. Rational approximation of frequency domain responses by vector fitting. *I*EEE Transactions on Power Delivery, 14(3):1052–1061, 1999.

[55] C. P. Coelho, J. Phillips, and L. M. Silveira. A convex programming approach for generating guaranteed passive approximations to tabulated frequency data. *I*EEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 23(2):293–301, 2006.

[56] G. Strang. *Linear algebra and its applications*. Thomson, Brooks/Cole, 2006.

[57] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *T*he Quarterly of Applied Mathematics, 9(1):17–29, 1951.

[58] N. Brisebarre, F. De Dinechin, and J. M. Muller. Integer and floating-point constant multipliers for FPGAs. In *A*SAP' 08: Proceedings of the 19^{th} IEEE International Conference on Application Specific Systems, Architectures and Processors, pages 239–244, 2008.

[59] P. K. Hanumolu, G. Y. Wei, and U. K. Moon. Equalizers for high-speed serial links. *I*nternational Journal of High Speed Electronics and Systems, 15(2):429–458, 2005.

[60] J. A. Davis and J. D. Meindl. *Interconnect technology and design for gigascale integration*. Springer, Netherlands, 2003.

[61] G. Balamurugan, B. Casper, J. E. Jaussi, M. Mansuri, F. O'Mahony, and J. Kennedy. Modelling and analysis of high-speed I/O links. *I*EEE Transactions on Advanced Packaging, 32(2):237–247, 2009.

[62] A. S. Sedra and K. C. Smith. *Microelectronic circuits*. Oxford University Press, 2007.

[63] L. T. Pillage and R. A. Rohrer. Asymptotic waveform evaluation for timing analysis. *I*EEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 9(4):352–366, 1990.

[64] K. J. Kerns, I. L. Wemple, and A. T. Yang. Stable and efficient reduction of substrate model networks using congruence transforms. In *I*CCAD '95: Proceedings of the IEEE/ACM International Conference on Computer-Aided design, pages 207–214, 1995.

[65] P. Feldmann and R. W. Freund. Efficient linear circuit analysis by Padé approximation via the Lanczos process. *I*EEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 14(5):639–649, 1995.

[66] L. Silveira, M. Kamon, I. Elfadel, and J. White. A coordinate-transformed Arnoldi algorithm for generating guaranteed stable reduced-order models of RLC circuits. *Computer Methods in Applied Mechanics and Engineering*, 169(3):377–389, 1999.

[67] A. Odabasioglu, M. Celik, and L. T. Pileggi. Prima: Passive reduced-order interconnect macromodeling algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):645–654, 1998.

[68] S. R. Nassif. Power grid analysis benchmarks. In *A*SPDAC '08: Proceedings of the 13^{th} IEEE/ACM Asia and South Pacific Design Automation Conference, pages 376–381, 2008.

[69] Download link for the IBM power grid benchmark set: `http://dropzone.tamu.edu/~pli/PGBench/`.

[70] A. V. Karthik, S. Ray, P. Nuzzo, A. Mishchenko, R. K. Brayton, and J. Roychowdhury. ABCD-NL: Approximating continuous non-linear dynamical systems using purely Boolean models for analog/mixed-signal verification. In *ASPDAC '14: Proceedings of the* 19^{th} *Asia and South Pacific Design Automation Conference*, pages 250–255, 2014.

[71] `http://www.analog.com/Analog_Root/static/techSupport/designTools/spiceModels/license/spice_general.html?cir=AD8079A.cir`.

[72] `http://ltspice.linear.com/software/LTC.zip`.

[73] A. Mishchenko, R. K. Brayton, S. Jang, and K. Chung. A power optimization toolbox for logic synthesis and mapping. 2009.

[74] D. T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *T*he Journal of Chemical Physics, 115(4):1716–1733, 2001.

[75] D. T. Gillespie and L. R. Petzold. Improved leap-size selection for accelerated stochastic simulation. *T*he Journal of Chemical Physics, 119(16):8229–8234, 2003.

[76] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a Scala embedded language. In *D*AC '12: Proceedings of the 49^{th} Design Automation Conference, pages 1216–1225, 2012.

[77] I. M. Filanovsky and H. Baltes. CMOS Schmitt trigger design. *I*EEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, 41(1):46–49, 1994.

[78] S. H. Hall and H. L. Heck. *Advanced signal integrity for high-speed digital designs*. John Wiley & Sons, 2011.

[79] B. K. Casper, M. Haycock, and R. Mooney. An accurate and efficient analysis method for multi-Gb/s chip-to-chip signaling schemes. In *Symposium on VLSI Circuits*, pages 54–57, 2002.

[80] J. E. Jaussi, G. Balamurugan, D. R. Johnson, B. K. Casper, A. Martin, J. Kennedy, N. Shanbhag, and R. Mooney. 8-Gb/s source-synchronous I/O link with adaptive receiver equalization, offset cancellation, and clock de-skew. *IEEE Journal of Solid-State Circuits*, 40(1):80–88, 2005.

[81] P. K. Hanumolu, G. Y. Wei, and U. K. Moon. Equalizers for high-speed serial links. *International Journal of High Speed Electronics and Systems*, 15(2):429–458, 2005.

[82] J. A. Davis and J. D. Meindl. *Interconnect technology and design for gigascale integration*. Springer, Netherlands, 2003.

[83] `http://download.intel.com/education/highered/signal/ELCT865/Class2_15_16_Peak_Distortion_Analysis.ppt`.

[84] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen. *Introduction to algorithms*. The MIT press, 2001.

[85] `http://en.wikipedia.org/wiki/8b/10b_encoding`.