FSM Model Abstraction for Analog/Mixed-Signal Circuits by Learning from I/O Trajectories

Chenjie Gu and Jaijeet Roychowdhury
{gcj,jr}@eecs.berkeley.edu
EECS Department, University of California, Berkeley

***Abstract*— Abstraction of circuits is desirable for faster simulation and high-level system verification. In this paper, we present an algorithm that derives a Mealy machine from differential equations of a circuit by learning input-output trajectories. The key idea is adapted from Angluin's DFA (deterministic finite automata) learning algorithm [1] that learns a DFA from another DFA. Several key components of Angluin's algorithm are modified so that it fits in our problem setting, and the modified algorithm also provides a reasonable partitioning of the continuous state space as a by-product. We validate our algorithm on a latch circuit and an integrator circuit, and demonstrate that the resulting FSMs inherit important behaviors of original circuits.**

I. Introduction

Circuit simulation and verification usually involve several levels of abstraction. This is crucial to design automation and large circuit designs. For examples, transistor-level modeling of circuits leads to SPICE simulation (solving differential algebraic equations of circuits) that is much faster than solving underlying PDEs of semiconductor devices. This abstraction makes SPICE continue being the golden standard for analog designers. At a higher level, RTL-level modeling of digital circuits dramatically speeds up simulations (using Verilog) and changes the design methodology, thus making VLSI designs (of millions of transistors) possible.

The mappings between different levels of abstraction are highly desirable. On the one hand, we often hope to extract high-level behaviors of a large circuit. This usually requires extensive simulations of the whole circuit. However, SPICE-level simulation of large circuits is inappropriate due to its extremely expensive computational cost. Therefore, low-level circuit models are replaced with their high-level much simpler models so that simulations can be finished within a reasonable amount of time. For example, mixed-signal designs (such as Sigma-Delta ADCs, phase locked loops, high-speed I/O links, *etc.*) usually involve close interactions between analog and digital sub-circuits. However, digital blocks (such as digital filters, phase-frequency detectors or frequency dividers) can consist of a large number of transistors – much larger than their analog counterparts. These digital circuits severely slow down the simulation, while simulation results may only capture unnecessarily accurate details of digital blocks. Since for digital circuits, an RTL representation, or even just a finite state machine model, can describe circuit behaviors well enough, it is preferable to use these high-level models during simulations of the whole system.

On the other hand, the *ideal* high-level block may not exhibit correct circuit behaviors in many situations. For example, in aggressive low-power and high-speed designs, the design margin is pushed to an extreme, and the smallest transistors are used which lead to large parameter variations that make circuit exhibit unexpected behaviors. For another example, in communication systems such as high-speed I/O links, in order to meet the power budget, some components may be intentionally designed to be erroneous when used stand-alone, but the errors can be corrected by the error-resilient mechanisms. These components may exhibit different erroneous behaviors according to different inputs. In both cases, we desire to derive a high-level model (such as an FSM) from the low-level model (such as a transistor-level netlist), hoping that the high-level model captures important details of the low-level model that affect circuit behaviors.

In this paper, we consider the problem of deriving an FSM model from a transistor-level circuit description. We present an algorithm whose key idea is based on an FSM learning algorithm (Angluin's DFA learning algorithm [1]) previously developed to learn an FSM from I/O traces of another FSM. However, we aim to learn an FSM (a discrete-time discrete-value dynamical system) from a circuit description (a continuous-time continuous-value nonlinear dynamical system). To do that, we have made considerable modifications of Angluin's algorithm to make it applicable in our problem.

As detailed in Section III, Angluin's algorithm is essentially a well-directed search for a DFA (deterministic finite automata, a special FSM) that matches all I/O traces (simulation trajectories) provided to the algorithm. However, in the circuit setting, a Mealy machine or a Moore machine is a more preferable model than a DFA since a Mealy/Moore machine directly describes how inputs and states affect the outputs of a circuit. By employing the similar key idea in Angluin's algorithm, we revise the original algorithm so that it directly extracts a Mealy machine.

Two other major modifications of the algorithm deal with two key subroutines in Angluin's algorithm, a simulator that generates input-output trajectories of the original FSM and an equivalence checker that checks the equivalence between the original and the learned FSM. We have provided substitutes for these two subroutines that work in our application. Since our adapted subroutines are based on SPICE simulation of the circuit, they are able to capture detailed nonlinear dynamics of the circuit.

With the FSM generated for a circuit, we further interpret the meaning of each discrete state in the resulting FSM. We establish connections between the FSM finite state space and the original continuous state space. As a by-product, we obtain a natural partitioning of the continuous state space, each region of which corresponds to a state in the FSM. While not discussed in great detail in this paper, this partitioning can be utilized in several other macromodeling/abstraction techniques, such as TPWL MOR techniques [2], [3] and hybrid system abstraction [4].

We demonstrate and validate our algorithm on two circuits. In the first example, we extract FSM models for an erroneous latch to be used in a high-speed I/O link circuit. By applying our algorithm, we obtain several FSMs of the latch with respect to different input swings, and we have verified that these FSM models are able to reproduce error behaviors observed in SPICE simulations. The second example is a non-ideal integrator circuit. We show from this example some important heuristics when applying our algorithm. The results are also validated against SPICE simulations, and the FSMs are shown to be good behavioral models of the integrator.

The rest of the paper is organized as follows. In Section II, we motivate and define the FSM model abstraction problem, and review some relevant work. In Section III, we review Angluin's DFA learning algorithm, and explain how we modify Angluin's algorithm to learn a Mealy machine from another Mealy machine. In Section IV, we provide two key subroutines that enables learning from a set of differential equations, and show connections between

978-1-4244-7514-8/11/$26.00 ©2011 IEEE

FSM states and the continuous state space. In Section V, we present experimental results of two circuit examples and verify that the FSM models we extract are able to reproduce qualitative behaviors of original circuits.

II. Problem Definition and Background

A. Problem Definition

We define the problem of *model abstraction* to be the extraction of a high-level (approximate) model from a low-level model.

Choices of low-level and high-level models differ according to applications. For example, the low-level model can be a SPICE netlist or the corresponding differential equation; the high-level model can be a small-signal linear model, a compact model, a smaller set of differential equations, or a graphical model such as a finite state machine or a dynamic Bayesian network. Specifically in this paper, the low-level and high-level models are circuit differential equations and a finite state machine model.

In the abstraction procedure, we require that the high-level model retains "important" behaviors of the low-level model so that it can serve as a substitute for the low-level model in high-level simulations and verifications. Here, we describe one criterion from an input-output perspective, and this will be used later in our algorithm.

Consider a circuit modeled by a set of differential equations

$$\begin{aligned} &\frac{d}{dt}\vec{q}(\vec{x}(t)) + \vec{f}(\vec{x}(t)) + \vec{B}\vec{u}(t) = 0, \\ &\vec{y}(t) = \vec{l}^T\vec{x}(t) + \vec{d}^T\vec{u}(t), \end{aligned} \tag{1}$$

where $\vec{x} \in \mathbb{R}^n$ are state variables (for example, node voltages in circuit equations), $\vec{u}$ are inputs (for example, voltage and current sources), and $\vec{y}$ are outputs. Therefore, the input-output pair of (1) is a pair of continuous-time continuous-value waveforms $(u(t), y(t))$. (For simplicity, we consider single-input single-output (SISO) systems, but it is straightforward to extend the idea to MIMO systems.)

Also consider a Mealy FSM defined by a 6-tuple $(Q, Q_0, \Sigma, \Lambda, \delta, G)$, where Q is a finite set of states, Q_0 is the initial state, Σ is a finite set of input alphabet, Λ is a finite set of output alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function that maps a pair of a state and an input symbol to the next state, and $G : Q \times \Sigma \rightarrow \Lambda$ is the output function that maps a pair of a state and an input symbol to an output symbol. Therefore, the input-output pair of an FSM is a pair of discrete-time discrete-value sequences $(u_i, y_i), i = 0, 1, 2, \cdots$, where $u_i \in \Sigma$ and $y_i \in \Lambda$.

To relate the FSM to differential equations (1), we define a unique mapping $t : \{u_i\} \mapsto u(t)$ where $u(ih) = u_i$, so that given an input sequence $\{u_i\}$ of the FSM, we can obtain a unique specification of input waveforms $u(t)$ for (1). Using $\{u_i\}$ and $u(t)$ as inputs to the FSM and (1) respectively, we can then obtain the outputs $\{y_i\}$ and $y(t)$. Similarly, we relate $y(t)$ to y_i by considering sampled points $y(ih)$ and by noticing that each output symbol corresponds to a region of outputs defined by a set of inequalities $A_i y < b_i$ (*e.g.*, "0" corresponds to $y \leq 0$ and "1" corresponds to $y > 0$).

Therefore, we say that an FSM is an abstraction of (1) if given a set of input sequences $\{u_i\}$ that correspond to input waveforms $u(t)$, the sampled outputs $\{y(ih)\}$ of (1) satisfy the inequalities of y_i, *i.e.*, $A_i y(ih) < b_i$. This notion of equivalence between a set of differential equations and an FSM is also visualized in Fig. 1.

This definition still requires us to choose the set of input sequences (*i.e.*, also known as the *training set* in supervised learning literatures [5]), the sampling time step h, and the mapping function $t : \{u_i\} \mapsto u(t)$. For the training set, we can use the inputs that are commonly encountered in applications such as square waveforms and sinusoidal waveforms. For the sampling time step h, it may be the clock period in clocked circuits, or an appropriate value that can capture system dynamics (*e.g.*, making the sampling frequency larger than the Nyquist rate). For the mapping function t, it is straightforward in

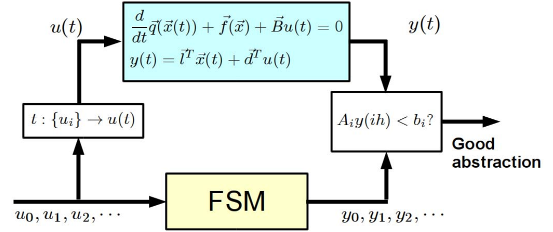


Fig. 1. Equivalence between an FSM and a set of differential equations.

digital applications since input waveforms basically switch between "0" and "1". In analog applications, we can implement the output of the mapping function to be a quantized piece-wise linear waveform.

B. A Motivating Example

We introduce a simplified latch model to illustrate the need for model abstraction and to demonstrate our approach in Section V. Latches are extensively used in mixed-signal designs, and deserve efforts to derive a good model.

Despite the fact that most latches are viewed as an ideal latch that has an ideal FSM representation shown in Fig. 2, we encounter non-ideal/erroneous latches in aggressive low-power designs due to large parameter variations and reduced signal voltages. In these cases, the corresponding FSM is no longer the one in Fig. 2, but more complicated in order to model the non-ideality.

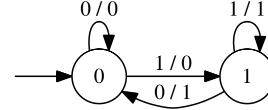


Fig. 2. The FSM of an ideal latch.

We consider a simplified latch circuit shown in Fig. 3. The latch is composed of a sampler (consisting of a multiplexer) and a regenerator (consisting of two cross-coupled inverters). Each output of the multiplexer and inverters is followed by an RC circuit that is not shown in Fig. 3. The latch samples the $DATA$ input when CLK is "1". When CLK is "0", two inverters are cross-coupled, and re-generate the output (QBB).

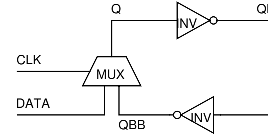


Fig. 3. A simplified latch model.

The differential equations for this latch are

$$\begin{aligned} RC\frac{dx_1}{dt} + (x_1 - f_{MUX}(u_1, u_2, x_3)) &= 0, \\ RC\frac{dx_2}{dt} + (x_2 - f_{INV}(x_1)) &= 0, \\ RC\frac{dx_3}{dt} + (x_3 - f_{INV}(x_2)) &= 0, \end{aligned} \tag{2}$$

where state variables x_1, x_2, x_3 are node voltages v_Q, v_{QB}, v_{QBB}, respectively; inputs u_1, u_2 are CLK and $DATA$, respectively; f_{INV} and f_{MUX} are I/O functions defined by

$$\begin{aligned} f_{INV}(x) &= -\tanh(10x), \\ f_{MUX}(C, A, B) &= \frac{A+B}{2} + \frac{B-A}{2}\tanh(10C), \end{aligned} \tag{3}$$

for the inverter and the multiplexer shown in Fig. 4.

One important erroneous behavior of this latch (also seen in practical CMOS latches [6]) is that when the input swing is reduced due to power budget, the latch needs several consecutive "1"s or "0"s at the input in order to make a successful transition. For example, Fig.

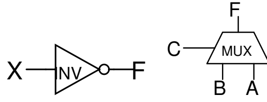


Fig. 4. An inverter and a multiplexer.

5 shows the result of a transient simulation of the simplified model when $V_{sw} = 1.6V$ and the input sequence is "010001100011100". When a single "1" is present at the input, it is not strong enough to turn the output to "1" at the end of the cycle. However, when two or more consecutive "1"s arrive, the output transits from "0" to "1". We show in Section V that the FSM we derive captures this behavior.

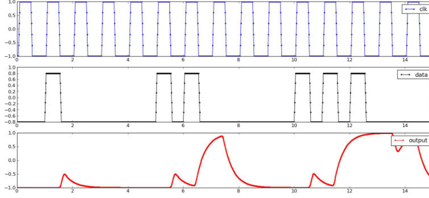


Fig. 5. Response of the latch when $V_{sw} = 1.6V$ and the input sequence is "10011000001001100000".

C. Relevant Work

There have been some studies on model abstraction in analog verification and hybrid system community. Many researches have been focusing on deriving models useful for verifying system properties such as safety property or probabilistic properties. This leads to the abstraction of non-deterministic FSMs [7] and probabilistic models such as dynamic Bayesian networks [8], [9]. These models are useful in verification, but have little use in simulation since they are not deterministic. The non-deterministicity may also be awkward in this situation since the original low-level model is a set of differential equations that have a unique solution, and are deterministic.

On the other hand, the main idea in most previous approaches is first choosing a good discretization/partitioning of the original continuous state space, and then building up a state machine based on this partitioning. This is indeed one of the main reasons that the resulting model is non-deterministic or probabilistic because under the same input, states within a region can move to several different neighboring regions. Besides, the explicit discretization in these methods also suffers from curse of dimensionality which restricts its application to only small-size problems.

In contrast, the FSM we derive is deterministic, and the algorithm avoids the curse of dimensionality caused by discretization of the continuous state space because it does not perform an explicit discretization. While this model may not be a completely sound model and may be inappropriate for formal verification, it is good for simulation, and is certainly useful if the input signals are well included in the training set.

III. Adaptation of Angluin's DFA Learning Algorithm to Learning Mealy Machines

In this section, we review Angluin's DFA learning algorithm, and show how the algorithm can be adapted to learn a Mealy machine from another Mealy machine.

A. Review of Angluin's DFA Learning Algorithm [1]

A DFA is a machine consisting of a 5-tuple: $(Q, \Sigma, \delta, Q_0, F)$, where Q is a finite set of states, Σ is a finite set of input alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, Q_0 is the start state, $F \subset Q$ is a set of accept states. Under the input sequence w, if the final state is in the set of accept states, the machine is said to accept w. Otherwise, it rejects w.

Given a DFA, Angluin's algorithm aims to learn the exact DFA from I/O traces of the DFA. The basic idea is to repeatedly conjecture machines according to I/O traces until an equivalent machine is obtained. The key data structure of the algorithm is an *observation table* which keeps track of necessary simulation results from which a machine can be conjectured.

An example of the observation table T is shown Table I, where λ means "empty", the set of row labels is called S and the set of column labels is called E, according to Angluin's notation. To fill in each cell of the table, we concatenate the row label with the column label, use that as the input sequence to the DFA to perform a membership query (simulation), and fill in the yes/no answer (to whether the DFA accepts this input sequence) in the cell. The observation table is further split into the top part and the bottom part, where row labels of the bottom part are obtained by concatenating row labels of the top part with all input alphabets. We will regard each row in the top part of the table as one state in the state machine, and therefore for each state in the top table, there must exist a row in the table that corresponds to its next state.

With the observation table, the key idea is then to differentiate different states by experiments, *i.e.*, we regard two rows as the same state if entries of the two rows are the same. For example, row "1" has the same entries as row "110" in Table I, and therefore represent the same state. However, to conjecture a DFA from an observation table, we need to place two constraints on the table:

1) The table is *closed*, *i.e.*, every row in the bottom part of the table has a corresponding row in the top part with identical results in all columns. This guarantees that none of the rows in the bottom part corresponds to a new state.
2) The table is *consistent*, *i.e.*, every pair of rows in the top part of the table with identical results in all columns also has identical results when any alphabet symbol is added. Mathematically, $\forall s_1, s_2 \in S$, if $\text{row}(s_1) = \text{row}(s_2)$, then $\forall a \in \Sigma$, $\text{row}(s_1 \cdot a) = \text{row}(s_2 \cdot a)$. This guarantees that any pair of states that we think are the same are still the same if the experiment string is extended by one input alphabet symbol.

The procedures to make the observation table closed and consistent are simple, and can be found in [1].

Given a closed and consistent observation table (S, E, T), we can then conjecture a DFA $(Q, \Sigma, \delta, Q_0, F)$ defined by

$$\begin{aligned} Q &= \{\text{row}(s) : s \in S \text{ of top table}\}, \\ Q_0 &= \text{row}(\lambda), \\ \delta(\text{row}(s), a) &= \text{row}(s \cdot a), a \in \Sigma, \\ F &= \{\text{row}(s) : T(s, \lambda) = 1\}. \end{aligned} \tag{4}$$

Using this DFA, we can feed it to an "equivalence checker" to check if the DFA is equivalent to the original one. If equivalence checker returns "yes", we terminate the algorithm. Otherwise, it returns an counter-example. In that case, we add the counter-example into the observation table, and repeat the procedure of conjecturing a new DFA until convergence.

Summarizing the above procedure, the Angluin's algorithm is sketched in Algorithm 1.

B. Learning Mealy Machines

As defined in Section II, a Mealy machine is similar to a DFA, but not exactly the same. It does not have accept states in DFA, but has an output function. This difference requires us to re-define the meaning of the observation table.

In Angluin's algorithm, each cell in the table is filled in by yes/no answers to whether the DFA accepts an input sequence. This can be viewed as the "output" of the DFA. Similarly, for Mealy

Algorithm 1 Angluin's DFA learning algorithm

1: Construct the initial observation table (S, E, T) with $S = \{\lambda\}$ and $E = \{\lambda\}$.
2: **repeat**
3: Make the observation table (S, E, T) closed and consistent.
4: Conjecture a DFA according to (4).
5: Check the equivalence between the conjectured DFA and the original DFA.
6: **if** not equivalent **then**
7: Add the counter-example in the observation table.
8: **end if**
9: **until** Equivalence checker returns "yes".

machines, we can fill in each cell in the table by the output of the Mealy machine. It turns out that using this new definition of the observation table, the concepts of closed-ness and consistence of the observation table still holds [10], and we can conjecture a Mealy machine $(Q, Q_0, \Sigma, \Lambda, \delta, G)$ from this new observation table by

$$\begin{aligned} Q &= \{\text{row}(s) : s \in S \text{ of top table}\}, \\ Q_0 &= \text{row}(\lambda), \\ \delta(\text{row}(s), a) &= \text{row}(s \cdot a), a \in \Sigma, \\ G(\text{row}(s), a) &= T(s, a), a \in \Sigma. \end{aligned} \quad (5)$$

Therefore, in order to learn a Mealy machine from another Mealy machine using Angluin's idea, we just need to use the new observation table, and replace step 4 in Algorithm 1 by "Conjecture a Mealy machine according to (5)".

IV. Learning Mealy Machines from Differential Equations

In this section, we describe adaptations of Angluin's algorithm to learn Mealy machines from differential equations. We show how we implement two key subroutines repeatedly called by Angluin's algorithm, and we interpret the states in the resulting Mealy machine in terms of regions in the original continuous state space. This connection also leads to a partitioning of the continuous state space.

In Angluin's algorithm, there are two subroutines that are repeatedly called:

1) The simulator. When building the observation table, the table entries are obtained by making membership queries (*i.e.*, is w accepted by the DFA?). In the case of learning Mealy machines, this corresponds to a simulation of the Mealy machine to obtain the output. Therefore, we need a simulator that can produce I/O traces of a machine/system.
2) The equivalence checker. The algorithm decides whether to terminate by making equivalence queries (*i.e.*, is the new machine M equivalent to the original machine?).

A. Implementation of the Simulator

The simulation of circuit differential equations is straightforwardly implemented by SPICE transient analysis. However, three problems remain to initiate the simulation and interpret the results:

1) To start the simulation, we need to specify an initial condition that corresponds to the initial state of the Mealy machine;
2) We need to transform the input sequence $\{u_i\}$ (to the FSM) to the input waveform $u(t)$ (to differential equations).
3) After the simulation, we need to translate the continuous output waveforms to discrete outputs of the FSM.

To determine the initial condition, we propose to use the DC solution of the circuit. This is usually an important state for circuits since circuits typically settle down to this state when the input is fixed (except for special cases like oscillators).

The transformation from the input sequence $\{u_i\}$ to to the input waveform $u(t)$ and the transformation from the output waveform $y(t)$ to output sequences $\{y_i\}$ are implemented by standard interpolation and quantization. However, one can define arbitrary transformations according to different applications. For example, the input sequence can specify phases of the input waveform at different time points, and the input waveforms are obtained by a transformation from the phase-domain to the voltage-domain. We can also specify binary values in the input sequence to be corresponding voltage levels of a digital square waveform during each clock cycle.

B. Implementation of the Equivalence Checker

While the equivalence between two FSMs are well-defined, the equivalence between an FSM and a continuous dynamical system is not entirely clear. Strictly speaking, a continuous dynamical system may not be represented by an FSM, since a continuous-time continuous-value system cannot be equivalent to a discrete-time discrete-value system. By sampling and quantizing the waveforms in continuous systems, however, there are situations that continuous dynamical system can be exactly represented by an FSM. The ideal latch is one such example.

In our case, we use the notion of equivalence defined in Section II. That is, we call the SPICE simulator to perform a set of simulations of circuit differential equations using $u(t)$ derived from $\{u_i\}$. If the sampled outputs $\{y(ih)\}$ lie in the corresponding quantization regions defined for $\{y_i\}$ (the output of the FSM to input sequence $\{u_i\}$), then we return "yes", *i.e.*, the FSM is equivalent to the differential equations. Otherwise, the input sequence is reported as an counter-example, and is added into the observation table to refine the FSM.

The training set used in the equivalence check is important. One common implementation is to use a Monte-Carlo sampling of input signals if *a priori* statistics of inputs are known. Another implementation useful in digital applications is to try out all digital sequences of a fixed length N. Although this leads to 2^N possible sequences, it is expected that for practical circuits with small number of I/O ports, short sequences are enough to characterize behaviors of the circuit. Furthermore, we can add/remove some special sequences into the training set to ensure some desirable properties of the system. For example, we may say that we want to capture results by applying five "1"s at the input, and therefore we add the input sequence of five "1"s into the training set. For another example, we may have a constraint on the input signal that it is not possible to have inputs of six or more consecutive "1"s (due to properties of the circuit generating the input signal), and then the training set can be pruned to speedup the equivalence checking procedure.

C. Connections between FSM States and the Continuous State Space

The Mealy machine extracted from differential equations has a finite set of states. Since the FSM is sometimes extracted in order to understand system behaviors, it poses a question of what these states actually mean in the original system.

The answer to this question traces back to one of the key ideas in Angluin's algorithm, *i.e.*, the algorithm differentiates different states by experiments. This idea leads to the fact that each row in the observation table represents a state in the FSM. On the other hand, each row is associated with a row label, and the initial state is associated with the row label λ. Suppose that a state corresponds to the row with row label s, then in order to tune the state of the FSM from the initial state to the state corresponding to row label s, we can apply the input sequence s to the FSM.

Based on this idea, we can therefore assign a point in the continuous state space for each state s in the FSM, by numerically solving differential equations using the continuous input waveform

converted from the input sequence s. Suppose the time-interval of the input is T_{input}, the solution of differential equations at time T_{input} then corresponds to a state in the FSM. Since the resulting FSM is supposed to be a good abstraction of the original system, transitions among these points are representative of system transitions/trajectories. However, they are still not satisfactory since discrete points in continuous state space have measure zero. It is therefore preferable to assign a region in the continuous state space to each state in the FSM.

To do this, we use the discrete points obtained above to construct a Voronoi tessellation/decomposition/diagram [11] which partitions a space with n points into convex polygons such that each polygon contains exactly one generating point and every point in a given polygon is closer to the generating point than to any other. These polygons, known as Voronoi polygons, are then associated with states in the FSM. As an example, a 3D Voronoi diagram with 4 points is shown in Fig. 7 in Section V.

Due to properties of the Voronoi tessellation, it can be viewed as a natural partitioning of the continuous state space given that the n points are obtained by definition of each conjectured state in the algorithm. With this partitioning, it is then possible to apply other techniques to further exploit low-level dynamics of the circuit. For example, we can construct simple nonlinear/linear systems for each partition, and this is very similar to ideas in hybrid system abstraction of continuous dynamical systems [4], and TPWL MOR techniques [2], [3].

Note that there is one key difference between this approach and many previous methods (described in Section II) on partitioning the continuous state space – while previous methods often suffer from curse of dimensionality, our approach avoids this problem as long as the original system can be represented by a small FSM. The number of partitions we obtain is the same as the size of the FSM, and we just need to store one point for each state to store the partitioning.

V. Validation

In this section, we apply our algorithm to a latch circuit and an integrator circuit. We derive Mealy machines for these circuits, and compare simulation results of Mealy machines to those of transient analysis. We show that the FSMs are able to capture qualitative behaviors of these circuits, and help to understand high-level properties of circuits.

A. Latch

The first example is the latch circuit introduced in Section II. As mentioned before, to obtain a power-performance trade-off, we hope to reduce the input swing as low as possible. However, as input swing is reduced, erroneous behaviors exhibit, and we want to extract these behaviors automatically.

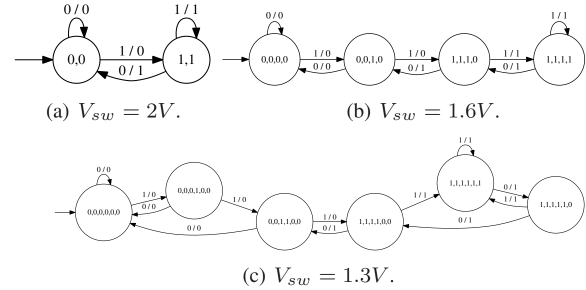
To apply our algorithm, we set the training set to be all input sequences of length 7 (7 clock cycles); the $t : \{u_i\} \to u(t)$ function is implemented to transform zero/one sequence to square waveforms of fixed duty cycle; the sampling time step h is chosen to be the clock cycle; and we set $y_i = 1$ if $y(ih) > 0$ and $y_i = 0$ otherwise.

Using this setting, we extract Mealy machines for this latch circuit with respect to input swing being $2V$, $1.6V$ and $1.3V$, as depicted in Fig. 6[1]. The numbers shown in each state are not arbitrary. They are values of rows in the observation table, *i.e.*, the quantized output with respect to corresponding input sequences. As an example, the observation table of the latch with input swing 0.8 is shown in Table I. We see that the state "0,0,0,0" corresponds to row λ, the state "0,0,1,0" corresponds to row "1", *etc.*. From Table I, we can derive the FSM in Fig. 6(b) using (5).

[1]Note that Mealy machines shown in Fig. 6 are indeed Moore machines, simply due to the fact that the output y is just the output voltage which corresponds to a state variable x_3. This is very common in circuit simulations, and a simple post-processing can be performed to obtain a Moore machine.

TABLE I
Observation Table for $V_{sw} = 1.6V$.

S \ E	0	1	10	00
λ	0	0	0	0
1	0	0	1	0
11	1	1	1	0
110	0	0	1	0
111	1	1	1	1
1110	1	1	1	0
11100	0	0	1	0
0	0	0	0	0
10	0	0	0	0
1100	0	0	0	0
1101	1	1	1	0
1111	1	1	1	1
11101	1	1	1	1
111000	0	0	0	0
111001	1	1	1	0



(a) $V_{sw} = 2V$.

(b) $V_{sw} = 1.6V$.

(c) $V_{sw} = 1.3V$.

Fig. 6. FSMs of the latch in Fig. 3.

Based on the Mealy machines in Fig. 6(a), we can claim that when input swing is $2V$, the latch behaves like an ideal latch; when input swing is $1.6V$, the input signal must persist for two consecutive cycles to avoid errors; when input swing is $1.3V$, the input signal must persist for three consecutive cycles to avoid errors. These properties can either be directly observed in simple FSMs, or be checked by model checking tools. To validate these claims, we also perform many SPICE simulations, and the same qualitative behaviors are observed.

Using these Mealy machines, we then construct Voronoi diagrams using discrete points corresponding to each state (Section IV-C). For example, the Voronoi diagram of the Mealy machine in Fig. 6(b) is shown in Fig. 7. The four points correspond to four states in Fig. 6(b), and the (blue) trajectories in Fig. 7 are simulation traces that leads to the discrete points of different states, and show typical transition dynamics of the latch. This partitioning of the state space also provides direct intuition on how circuit behave : the two states at two corners represent solid "0" and solid "1" states, and the two states in the middle capture dynamics in the metastable region, and can be viewed as weak "0" and weak "1" states.

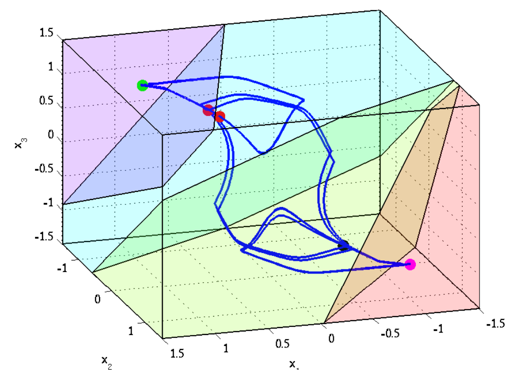


Fig. 7. Interpretation of states in the FSM ($V_{sw} = 1.6V$).

B. Integrator

The second example is an integrator circuit shown in Fig. 8, where $R = 1$, $C = 1$, and the opamp is approximated by a controlled voltage source which clips at $\pm 1V$ and has gain 10 and resistance 0.1. The input is a binary sequence where "1" and "0" are at voltage level $+V_{in}$ and $-V_{in}$, respectively, and each bit lasts for 1 second.

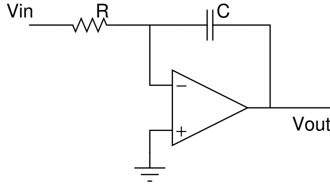


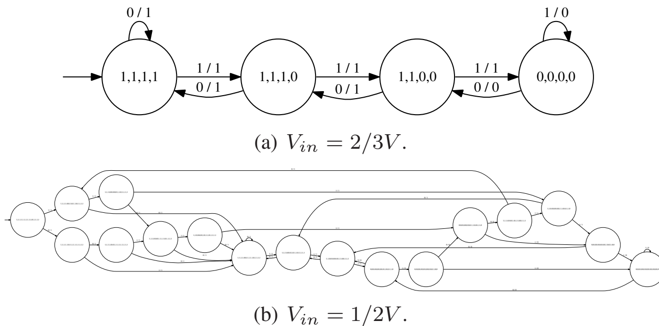
Fig. 8. Circuit diagram of an integrator circuit.

To initiate the algorithm, the step size h is chosen to be 1 second, *i.e.*, the period for each bit. The training set are all input sequences of length 8. The output is quantized by setting $y_i = 1$ when $y(ih) > 0$ and $y_i = 0$ otherwise. The initial state is chosen as the DC solution when the input is at the "0" level.

We first extract the FSM when the input level $V_{in} = 2/3V$. As shown in Fig. 9(a), the resulting FSM has 4 states. The "1,1,1,1" state corresponds to the state where the output is saturated at $1V$ and the "0,0,0,0" state corresponds to the state where the output is saturated at $-1V$. From this state machine, we immediately see how the integrator charges and discharges as different inputs are applied.

We stress that the initial condition/state in the learning algorithm is extremely important. Choosing a different initial condition may lead to FSMs with unnecessarily more states. For example, if the initial state is at a strange point which is almost never visited by the normally operating circuit, then this state is not representative of original circuit trajectories, and it makes the algorithm harder to explore important states by performing experiments. The heuristic we have used is to choose the initial state to be a DC solution. The DC state is always important in almost all circuits, and therefore is always good to be modeled in the FSM. Experiments also show that this heuristic leads to better FSMs (FSMs with less states and meaningful transitions).

However, in cases where the exact DC solution is not available or a random initial condition is chosen, we also have a simple heuristic to reduce the number of transient states in the FSM. The key idea is that since some transient states are never visited from other states, we may well discard these states. Equivalently, we should look for the strongly connected components of the DAG (Directed Acyclic Graph) of the FSM, and this typically filters out a lot of transient states. Therefore, to discard useless transient states, we just need to apply algorithms for finding strongly connected components, and fortunately, linear-time (with respect to the number of states) are available, such as [12].



(a) $V_{in} = 2/3V$.

(b) $V_{in} = 1/2V$.

Fig. 9. FSMs of the integrator circuit.

We also extract the FSM of the integrator when the input level $V_{in} = 1/2V$. The FSM is shown in Fig. 9(b). (Since the number of states is relatively large, the labels are not shown clearly.) This state machine, however, is much more complicated than what we would imagine of an ideal integrator – we may think that this machine should have 5 states that are connected like the one in Fig. 9(a). This wrong intuition stems from the fact that the circuit is not an ideal integrator – the amount output increases/decreases depends not only on the input, but also on the current state, and this behavior is exhibited when the input level is relatively small. To capture the non-idealities, more states are used by the algorithm.

While the FSM in Fig. 9(b) does not give direct intuition to designers, it does capture qualitative dynamics of the integrator. Therefore, we can simply apply model checking techniques to this FSM to check high-level properties of the system. For example, we may check "whether there exists an input sequence of length 4 so that we can change the output of the integrator from $-1V$ to $1V$". When the FSM is a good approximation of the continuous system, we can trust the result returned by the model checker, or we can use that result as a guidance to check circuit properties by simulations.

VI. Conclusion

FSM model abstraction of analog/mixed-signal circuits is desirable for high-level system simulation and verification. In this paper, we present an algorithm, based on ideas of Angluin's DFA learning algorithm, that automatically extracts a Mealy machine from circuit differential equations according to simulation trajectories. We validate the algorithm using two illustrative circuits. The extracted FSMs are able to reproduce important behaviors of original circuits, and can be used to check other circuit properties.

Acknowledgment

Chenjie Gu would like to thank Wenchao Li for the reference to [1], Baruch Sterin for many useful discussions, and the anonymous reviewers for corrections and suggestions.

References

[1] Dana Angluin. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[2] Michal Rewienski and Jacob White. A Trajectory Piecewise-Linear Approach to Model Order Reduction and Fast Simulation of Nonlinear Circuits and Micromachined Devices. *IEEE Transactions on Computer-Aided Design*, 22(2), February 2003.

[3] Chenjie Gu and Jaijeet Roychowdhury. Model Reduction via Projection onto Nonlinear Manifolds, with Applications to Analog Circuits and Biochemical Systems. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 85–92, Piscataway, NJ, USA, 2008. IEEE Press.

[4] George J. Pappas. *Hybrid Systems: Computation and Abstraction*. PhD thesis, EECS Department, University of California, Berkeley, 1998.

[5] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, corrected edition, August 2003.

[6] Naresh Shanbhag and Elyse Rosenbaum. Personal communication.

[7] Walter Hartong, Lars Hedrich, and Erich Barke. On discrete modeling and model checking for nonlinear analog systems. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 401–413, London, UK, 2002. Springer-Verlag.

[8] Kevin Patrick Murphy. Dynamic bayesian networks: Representation, inference and learning, 2002.

[9] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[10] Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of components guided by incremental state machine learning. In *TAIC-PART '06: Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, pages 59–70, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000.

[12] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121 – 123, 1979.