DAE2FSM: Automatic Generation of Accurate Discrete-Time Logical Abstractions for Continuous-Time Circuit Dynamics

Karthik .V. Aadithya*‡ and Jaijeet Roychowdhury*
*Department of Electrical Engineering and Computer Science, The University of California, Berkeley, CA, USA
‡Contact author. Email: aadithya@berkeley.edu

ABSTRACT

We abstract the I/O functionality of continuous-time dynamical systems (e.g., SPICE netlists with combinational and sequential logic) as 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 it to learn high-fidelity abstractions of multi-level-discretized, multi-input/multi-output systems. Our approach, 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 our technique to produce multi-level-discretized FSM representations of digital systems that nevertheless 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 simulations). Finally, we demonstrate that our technique 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 (BSIM4), 0-to-5 increment/decrement counter.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques] State Diagrams
B.2.2 [Performance Analysis and Design Aids] Simulation

General Terms

Algorithms, Design, Theory

Keywords

Finite State Machine Learning, Circuit Simulation

1. INTRODUCTION

With technology scaling to 22nm and below, individual devices are increasingly becoming non-ideal, thus compromising the clean Boolean abstractions that underpin the effectiveness and power of the digital design paradigm. Indeed, 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 nonlinear analog dynamics, analog waveshapes, noise/interference, etc., compounded by increased variability, cannot be directly captured within the Boolean modelling and simple delay frameworks that are natural for digital systems.

Purely SPICE-level simulation-based approaches for validation and debugging are impractical, on account of the large sizes of typical digital systems. In existing design methodologies, components in cell libraries are simulated extensively at the SPICE-level, over a range of PVT corners, to verify functionality and to characterize delays. This process does not, however, provide executable abstractions (such as finite state machines) that can reproduce details of analog wave-shapes; nor is it suited for components whose functionality is affected significantly by complex analog effects.

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

In this work, we develop and demonstrate techniques (collectively dubbed DAE2FSM[1]) to abstract executable Boolean descriptions of transistor-level circuits. The central notion of DAE2FSM is to approximate transistor-level circuits *accurately* as finite state machines (FSMs) by adapting and applying computational machine learning techniques. The resulting FSMs can not only capture the intended logical functionality of the circuit being abstracted, but also take into account analog effects and non-idealities, producing "non-ideal FSMs" that accurately reflect actual (rather than intended) operation (see Fig. 1).

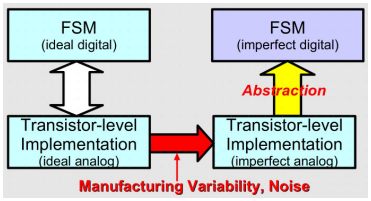


Figure 1: DAE2FSM: Transistor level non-idealities are captured in FSM representations.

We develop and apply an Angluin-based [1] computational learning technique that uses *finite alphabets of* $N \geq 2$ *symbols*, thus moving beyond the binary symbols used in prior work [2, 3]. 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. We demonstrate the application of these advances on transistor-level latch and flip-flop circuits, as well as on a counter circuit composed of several sequential and combinational components.

DAE2FSM features several points of novelty and promise. FSMs generated by DAE2FSMcan 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.

An important feature of DAE2FSM is that it is suitable for application to practical industrial circuits. Detailed, non-linear SPICE-level circuits can be utilized unchanged, within simulation environments of the user's choice, ensuring that no analog subtlety that SPICE can predict is ignored. At the same time, 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 a system perspective) are captured; in other words, only what is needed is represented in the learned FSM. Indeed, the automated, push-button nature of FSM generation frees the user from understanding in detail how a given transistor-level block functions; all that is needed is a SPICE-level netlist that simulates. 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, is very attractive.

Moreover, by representing transistor-level circuits as FSMs, DAE2FSM opens the door to the application of *Boolean formal verification and model checking* techniques (e.g., [4, 5]) for determining whether a given transistor-level component with strong analog characteristics satisfies a given property. Unlike alternative formal approaches based on hybrid systems (e.g., [6–9]), DAE2FSM neither suffers from scalability limitations, nor requires *a priori* modelling simplifications of the SPICE-level transistor blocks involved.

The rest of the paper is organized as follows. In §2, we outline the multi-symbol learning technique underlying DAE2FSM. In §3, we present detailed results 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.

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

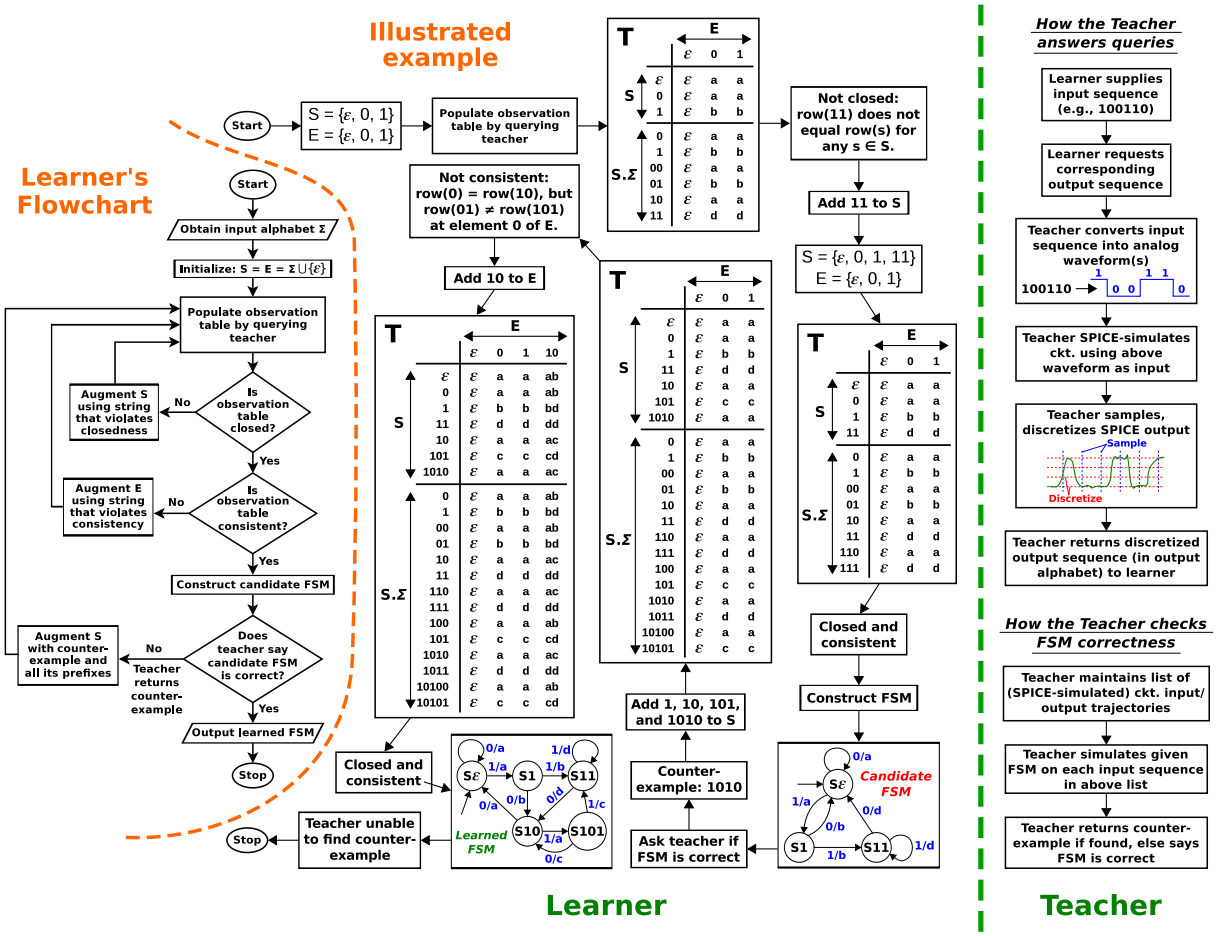


Figure 2: Flowchart and an example that describe our technique for fully automated multi-symbol Mealy machine learning.

2. CORE TECHNIQUE: MULTI-SYMBOL ANGLUIN-STYLE MEALY MACHINE LEARNING

We now describe our core technique for abstracting a given SPICE netlist as a finite state, multi-symbol Mealy machine[2]. Our approach derives from the well-known Angluin's algorithm [1] for Deterministic Finite Automata (DFAs)[3], which we have modified to learn multi-symbol Mealy machine abstractions of continuous dynamical systems such as circuits.

Our FSM abstraction algorithm for circuits has two key entities: a teacher, and a learner. Before the learning begins, both entities decide on an input alphabet Σ, and an output alphabet Γ (these can be any finite sets of symbols).

The teacher has access to the SPICE-level circuit netlist and a SPICE simulator. The learner, on the other hand, operates purely in the discrete domain: it has no knowledge of the circuit, and the only way it learns about the circuit is by asking specific questions of the teacher. These questions can take two forms: (a) I/O queries, and (b) FSM checks.

An I/O query involves the learner presenting the teacher with an input sequence (any word from the input alphabet), to which the teacher responds with an output sequence (a word from the output alphabet of length equal to the input sequence). Given the input sequence, the teacher constructs an input *waveform* from it (as shown in Fig. 2 (right)), and then SPICE-simulates the circuit on this input waveform. The teacher then discretizes the SPICE output into symbols from the output alphabet, and returns the discretized sequence as the "answer" to the learner's query. Thus, with each I/O query, the learner increases its knowledge about the circuit; eventually, it learns enough about the circuit to propose an FSM abstraction (in the form of a multi-symbol Mealy machine) to the teacher. This is the second type of query (an FSM check). Given an FSM proposed by the learner, the teacher runs a number of simulations comparing the outputs produced by the FSM with discretized versions of outputs produced by SPICE-simulating the circuit. If the teacher finds a discrepancy between the FSM and SPICE, it alerts the learner to the counter-example, and the learner uses this information to refine its FSM. If the teacher is unable to find a counter-example, it accepts the FSM as a faithful discrete-domain representation of the continuous-domain circuit behaviour.

We note that the teacher has enormous freedom in the way it converts an input *sequence* into an input *waveform*: for example, the teacher can interpret input symbols as *quantized voltage levels* (e.g., symbol "a" means 0V, symbol "b" means 0.1V, etc.), leading to a *multi-level-discretized* learned FSM. Alternatively, the teacher can interpret input symbols as *bit vectors* specifying boolean values for multiple circuit inputs (e.g., symbol "a" means 000, symbol "b" means 001,etc.); this interpretation results in a *multi-input* Mealy machine abstraction for the circuit. Another possibility is that the teacher can interpret the input symbols as *switching events* (e.g., symbol "a" means the clock switches, symbol "b" means 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 §3.2 and §3.3). 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 (§3.4), a multi-level FSM would 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 would be best-suited (see §3.5).

Having presented the teacher's side, we now discuss the learner's algorithm (shown, along with an example, in Fig. 2 (left)). At any point, the learner maintains two sets of words over the input alphabet, S and E. In addition, it maintains an *observation table* T that contains all the information acquired about the circuit thus far. Initially, $S = E = \Sigma \cup \varepsilon$, where ε is the empty string. At any time, for every ordered pair $(s,e) \in (S \cup S.\Sigma) \times E$ (where . denotes concatenation), the observation table T contains an entry[4] $T(s,e)$, which is equal to the last $|e|$ output symbols returned by the teacher for the input sequence $s.e$ (Fig. 2 illustrates how T evolves as the algorithm progresses). For each $s \in S \cup S.\Sigma$, let row(s) denote the row corresponding to s in T.

[2]Recall that Mealy machines are FSMs that take in an input sequence, and produce an output symbol at each state transition, thereby returning an output sequence of length equal to the input sequence.

[3]Recall that DFAs are FSMs that take in an input string, and either output a "1" (indicating that the string has been accepted) or output a "0" (indicating that the string has been rejected).

[4]In Angluin's original algorithm for DFAs, T only contained 0/1 entries. However, to extend the algorithm to multi-symbol Mealy machine learning, we store output strings in T instead of binary values.

Using the information in T, the learner tries to match an *FSM state* to each word in S; for each $s \in S$, the learner associates with s the *final* state reached by the FSM on input 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 $\text{row}(s_1) = \text{row}(s_2)$. 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 recompleted.

Consistency: For every $s_1, s_2 \in S$ such that $\text{row}(s_1) = \text{row}(s_2)$, it should also be true that $\text{row}(s_1.\sigma) = \text{row}(s_2.\sigma)$ for every $\sigma \in \Sigma$. The intuition is that: one cannot associate identical FSM states with two different 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 the string $\sigma.e$ for which $T(s_1.\sigma.e) \neq T(s_2.\sigma.e)$ must be added to E, and T recompleted.

Thus, the learner issues I/O queries to the teacher until T is both closed and consistent. At that point, the learner proposes an FSM (whose states are associated with strings in S). This is repeated until the teacher is unable to find a counter-example to the learner's FSM. In Fig. 2 (left), 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 §3.4).

3. RESULTS

As mentioned earlier, we have applied the techniques of §2 to generate multi-symbol Mealy machine abstractions of latches, flip-flops, and circuits constructed from them. Here, we discuss these results in detail.

We begin by generating binary FSM abstractions of correctly functioning latches and flip-flops (§3.1), 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 (§3.2, §3.3); 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 §3.4, 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 §3.5, 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.

3.1 Binary FSMs for correctly functioning latches and flip-flops

We start with the simplest 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.

At this stage, the target FSMs are limited 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. 3 (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).

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

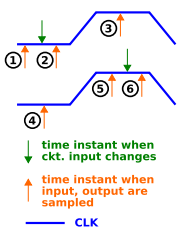


Figure 3: Binary FSMs learned for latches and flip-flops in various operating modes.

This gives rise to six possible timing scenarios, tabulated in Fig. 3. For each scenario, we used DAE2FSM to learn FSMs for both a D-latch and a D-flip-flop[5]. Fig. 3 shows that, depending on the scenario and the circuit, the learned FSM can be either a *buffer* or a *delay*. These FSMs are shown in Figs. 4 (a) and 4 (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. 3).

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).

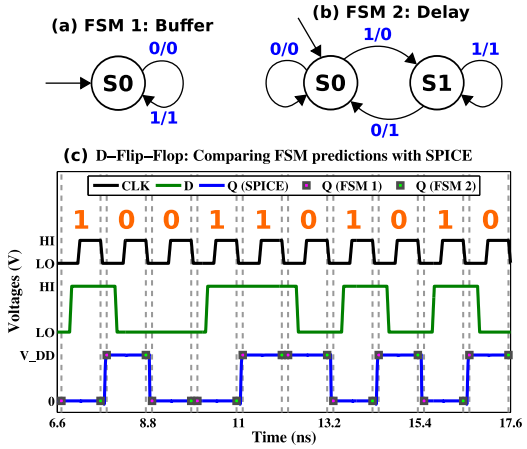


Figure 4: The *buffer* and *delay* FSMs returned by DAE2FSM accurately reflect the behaviour of an ideal D-flip-flop.

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 (c) shows a SPICE simulation[6] 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. 3, 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 (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 0V, and the output symbol "1" with the voltage V_{DD} (which, in this case, is 0.8V). 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 (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) the analog time series version of the delay FSM's predictions (which we know, from Fig. 3, to be valid just *before* the negative edge of the clock). From the figure, we see that this set of predictions also closely match the SPICE waveform at these time points (the grey vertical lines).

3.2 Multi-input FSMs for correctly functioning latches

The FSMs of the previous subsection 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. 3). 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 §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

[5]We note that a D-latch FSM has also been learned by the authors of [2]. However, that work considers only one of the scenarios outlined in Fig. 3, whereas we have meticulously analysed all possible scenarios. Also, the authors of [2] consider only latches, whereas we have developed FSMs for latches, flip-flops and beyond.

[6]All SPICE simulations in this paper have been carried out with 22nm devices, using BSIM4 device models. We obtained device parameters from [10], and the SPICE engine from [11].

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 §2 and in §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 §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.[7] 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. 5 (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. Indeed, as Fig. 5 (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. 5 (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. 5 (b) and 5 (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. 5 (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[8]; 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. 5 (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. 5 (b) and 5 (d) to arrive at a non-deterministic FSM. The rationale is that CLK in Fig. 5 (b) switches ahead of D at input symbol z, while Fig. 5 (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. 5 (f), whose non-z transitions are identical to the original FSMs, but whose z-transitions include non-determinism.

3.3 Multi-input FSMs for correctly functioning flip-flops

We now repeat the analysis of the previous subsection, but for a D-flip-flop[9] instead of a D-latch. The results (Fig. 6) roughly mirror those of the previous subsection (Fig. 5); 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 before). The auto-generated FSMs are shown in Figs. 6 (b) and 6 (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. 6 (b) and 6 (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. 6 (a) and 6 (c), the predictions made by these FSMs match very well with values sampled from SPICE simulations.

Finally, Fig. 6 (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. 6 (f).

3.4 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 role 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 [2]. In this work, we focus on *multi-level discretizations* applied to flip-flops and demonstrate how DAE2FSM captures novel failure modes.

Fig. 7 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. 7 (a)) can be adequately captured by a binary FSM, whereas the other (Fig. 7 (c)) needs a multi-level output alphabet (supported only by the multi-symbol approach).

In the first mode of failure[10] (Fig. 7 (a)), a single "1" applied at the input of the flip-flop 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. 7 (a). As shown in Fig. 7 (b), DAE2FSM is able to capture this failure mode. The FSM in Fig. 7 (b) starts at the "zero" state marked Sε. 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. 7 (b) is adequate to capture this failure mode.

The second failure mode, illustrated in Fig. 7 (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

[7] 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. 5 (f)).

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

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

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

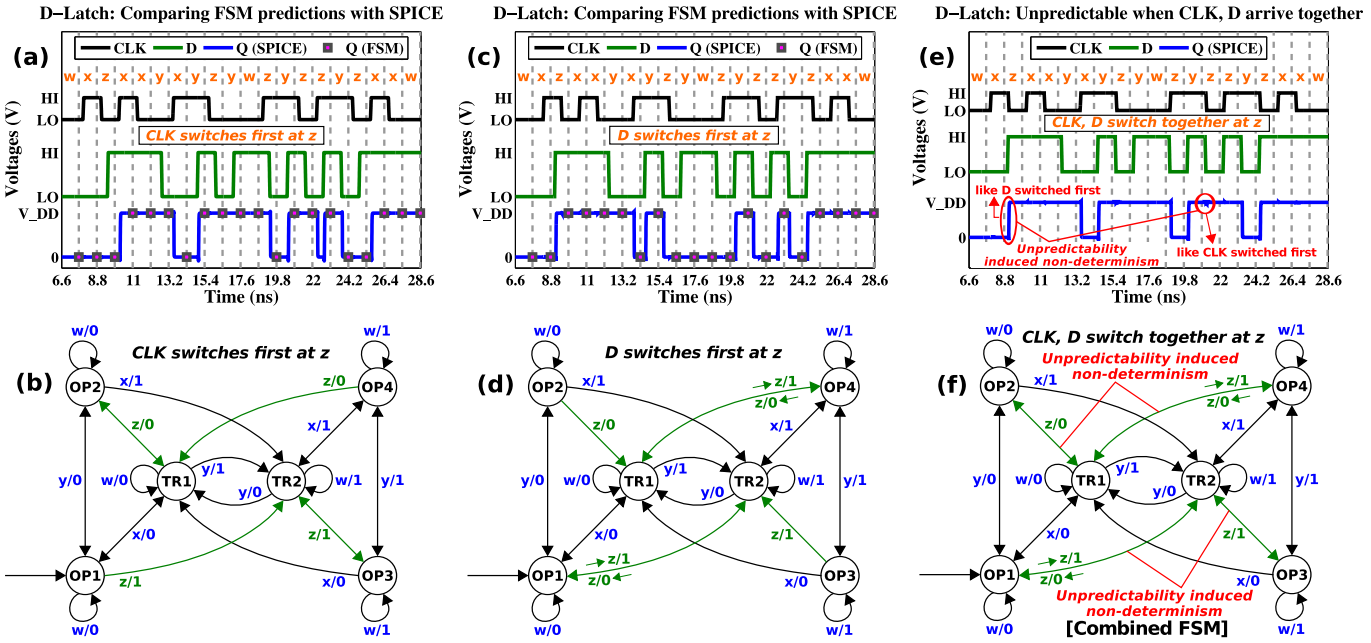


Figure 5: Multi-input DAE2FSM applied to construct multi-input FSM abstractions of a D-Latch.

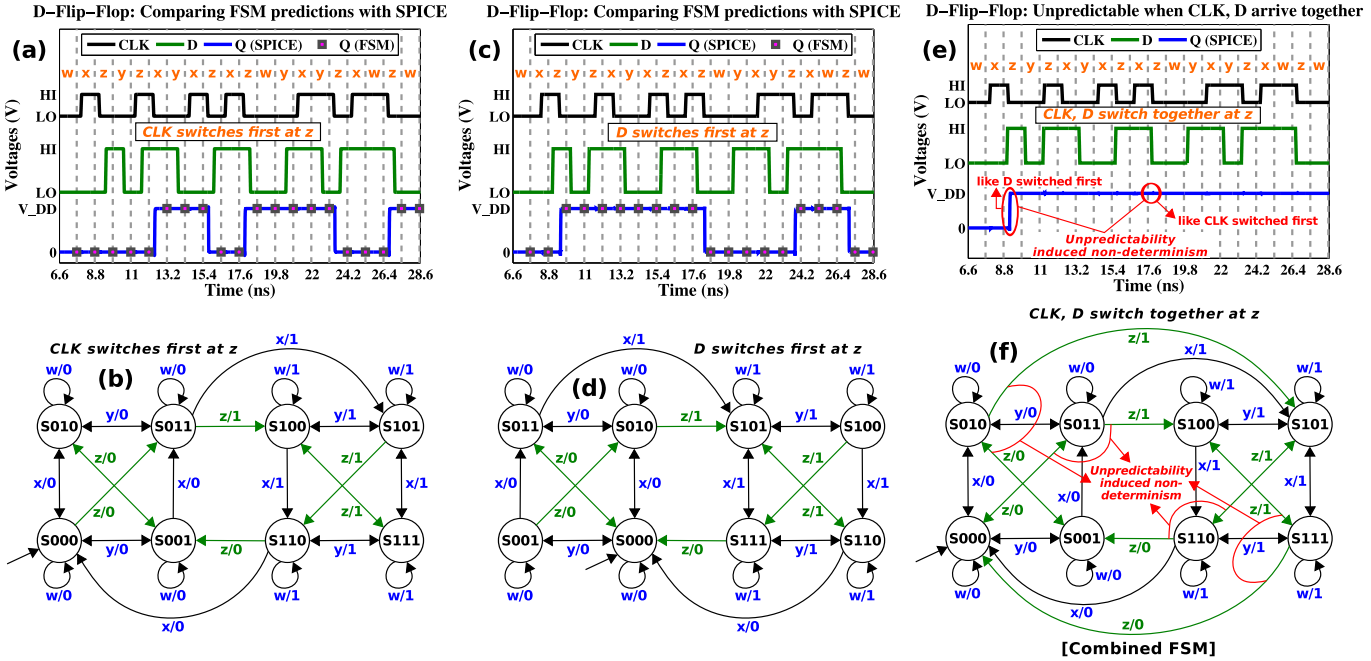


Figure 6: Multi-symbol DAE2FSM applied to construct unified FSM abstractions of a D-flip-flop.

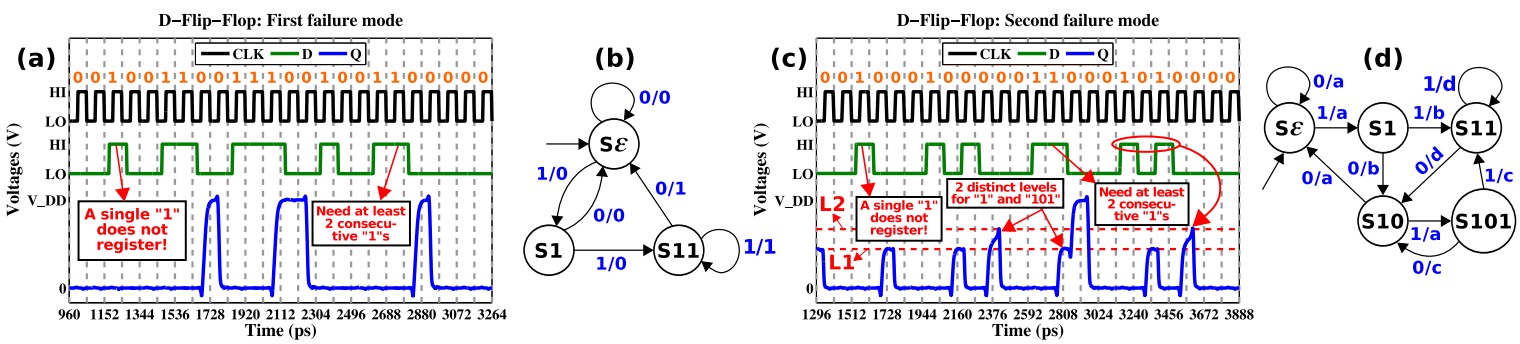
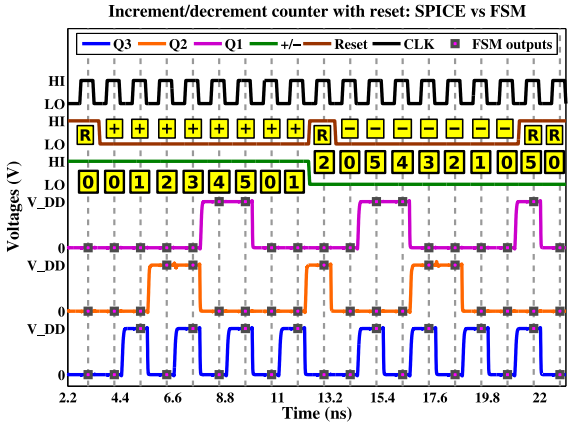


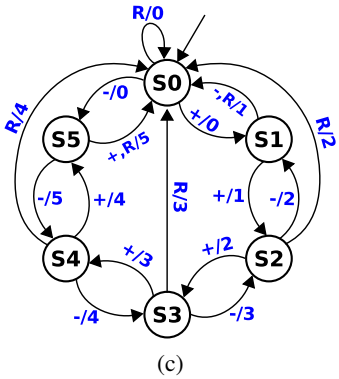
Figure 7: Failure modes observed for flip-flops, and FSMs learned for failing flip-flops.

Inputs		Action	Current state			Next state		
X	R		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 9: (Left) Table showing the state transitions of a 0-to-5 increment/decrement counter with reset (please see Fig. 8 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. (Right) Multi-symbol Mealy machine automatically learned by DAE2FSM for the counter. (Middle) 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.

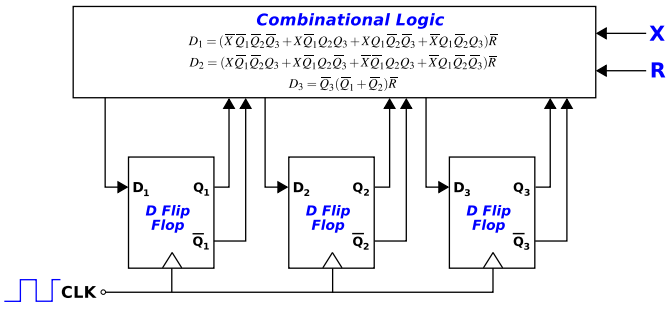


Figure 8: Schematic of a 0-to-5 increment/decrement counter with reset (please see Fig. 9 (a) for the corresponding state transition table).

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. 7 (d). The example of Fig. 2 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) for a "0" at the input, and "d" (the symbol for V_{DD}) 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.

3.5 A combinational/sequential, multi-input/multi-output case

Having demonstrated FSM abstraction 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. 8), which takes two (digital) inputs (not counting `CLK`) X and R, and returns three (digital) outputs Q_1, Q_2 and Q_3. The output bits Q_1 to Q_3 encode a whole number (the count) in the range 0-to-5 (0 and 5 included), with Q_1 (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. 9 (a) shows the state transition table associated with the counter, while Fig. 9 (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 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[11], respectively, for multi-symbol Angluin-based learning (see §2).

Fig. 9 (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. 9 (b)).

4. SUMMARY, CONCLUSIONS AND FUTURE WORK

In this paper, we have demonstrated DAE2FSM, a technique for automatically learning multi-symbol discrete-domain FSM abstractions of continuous-domain dynamical systems such as circuits. We have extended Angluin's algorithm to multi-symbol Mealy machine learning, which enables 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 our technique 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.

In future, we would like to extend our framework to automatically learn non-deterministic FSMs that characterise circuit behaviour for inputs that create race conditions. Closely related to this, we would also like to auto-generate FSMs that capture the effects of metastability on latches and flip-flops.

5. REFERENCES

[1] D. Angluin. Learning regular sets from queries and counter-examples. *Information and Computation*, 75:87–106, November 1987.

[2] C. Gu and J. Roychowdhury. FSM model abstraction for analog/mixed-signal circuits by learning from I/O trajectories. In *ASP-DAC '11: Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pages 7–12, 2011.

[3] C. Gu. *Model order reduction of non-linear dynamical systems*. PhD thesis, The University of California, Berkeley, 2011.

[4] E. M. Clarke, T. A. Henzinger, and H. Veith, editors. *Handbook of model checking*. Springer-Verlag, 2011.

[5] R. K. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV '10: Proceedings of the 22nd International Conference on Computer Aided Verification*, pages 24–40, 2010.

[6] R. Alur, T. A. Henzinger, G. Laferriere, and G. Pappas. Discrete abstractions of hybrid systems. *Proceedings of the IEEE*, 88(7):971–984, July 2000.

[7] C. L. Guernic and A. Girard. Reachability analysis of hybrid systems using support functions. In *CAV '09: Proceedings of the 21st International Conference on Computer Aided Verification*, pages 540–554, 2009.

[8] A. Girard, C. L. Guernic, and O. Maler. Efficient computation of reachable sets of linear time-invariant systems with inputs. In *HSCC '06: Proceedings of the 9th International Conference on Hybrid Systems: Computation and Control*, pages 257–271, 2006.

[9] C. Tomlin, I. Mitchell, A. M. Bayen, and M. Oishi. Computational techniques for the verification of hybrid systems. *Proceedings of the IEEE*, 91(7):986–1001, 2003.

[10] `http://ptm.asu.edu/modelcard/HP/22nm_HP.pm`.

[11] `http://www.spiceopus.si/`.

[11] 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.