

# Symbolic Model Checking of Analog/Mixed-Signal Circuits \*

David Walter, Scott Little, Nicholas Seegmiller, Chris J. Myers  
University of Utah  
Salt Lake City, UT 84112  
{dwalter, little, seegmill, myers}@vlsigroup.ece.utah.edu

Tomohiro Yoneda  
National Institute of Informatics  
Tokyo, Japan  
yoneda@nii.ac.jp

**Abstract—** This paper presents a Boolean based symbolic model checking algorithm for the verification of *analog/mixed-signal* (AMS) circuits. The systems are modeled in *VHDL-AMS*, a hardware description language for AMS circuits. The *VHDL-AMS* description is compiled into *labeled hybrid Petri nets* (LHPNs) in which analog values are modeled as continuous variables that can change at rates in a bounded range and digital values are modeled using Boolean signals. System properties are specified as temporal logic formulas using *timed CTL* (TCTL). The verification proceeds over the structure of the formula and maps separation predicates to Boolean variables. The state space is thus represented as a Boolean function using a *binary decision diagram* (BDD) and the verification algorithm relies on the efficient use of BDD operations.

## I. INTRODUCTION

While taking up only a small portion of the chip area, *analog/mixed-signal* (AMS) circuits are responsible for 50 percent of the errors that result in a redesign [11]. Therefore, improvements in AMS circuit validation methodology are very important. Analog circuit validation is typically achieved using SPICE simulation. Although mixed-signal validation can be done using *VHDL-AMS* simulation, it is often done in a more ad hoc way. Until recently, digital circuit validation also utilized this simulation-only based methodology, but now formal verification is often employed. Formal verification utilizes nondeterminism and state space exploration to simultaneously validate all possible simulations over a range of parameters and initial conditions. While simulation has the potential to identify any error, it is necessary to identify the particular simulation parameters that would result in each error. Formal verification approaches alleviate this necessity. These techniques, therefore, provide a promising mechanism to validate designs in the face of noise and uncertain parameters.

Perhaps the first work in the formal verification of AMS circuits is from Kurshan and McMillan in which analog circuit models are translated to finite state models using homomorphic transformations [12]. Hartong et al. verify analog circuits by dividing the continuous state space into regions that are represented in a Boolean manner [8]. This allows them to perform model checking using standard Boolean-based approaches though at some loss of accuracy. Tools for verifying hybrid systems have also been adapted to verify AMS circuits. Gupta et al. utilize *CheckMate* to verify analog circuits such

as a tunnel diode oscillator and a delta-sigma modulator [7]. In [3], Dang et al. use  $d/dt$  to verify a biquad low-pass filter. In [6], Frehse et al. use PHAVer to verify analog oscillator circuits. These approaches, however, require a user to describe an AMS circuit using a *hybrid automaton* which is unfamiliar to most AMS circuit designers. In [13], Little et al. adapt a zone-based algorithm for the verification of AMS circuits. This method, however, only supports constant rates of change for the continuous variables and conservatively abstracts the continuous state space.

This paper describes a new exact symbolic model checking algorithm for the verification of AMS circuits which supports ranges on the rates of change for the continuous variables. Figure 1 presents a flowchart of the steps in this verification method. A model of the AMS circuit is first specified by the designer using a subset of *VHDL-AMS* described below. By allowing the designer to specify the model in a language that is familiar to them, we hope to encourage the acceptance of formal verification methodologies. The *VHDL-AMS* description is automatically compiled into a *labeled hybrid Petri net* (LHPN) which includes Boolean signals to represent digital circuitry and continuous variables to model voltages and currents in the analog circuitry. The LHPN model provides a formalism for reasoning about the system being analyzed. System properties are specified as temporal logic formulas using *timed CTL* (TCTL). The TCTL can be automatically generated from **assert** statements in *VHDL-AMS* or more complicated properties can be specified by the designer. In [17], Seshia and Bryant describe a symbolic model checking procedure for real-time systems based on the one described in [9]. Their method maps separation predicates to Boolean variables so that analysis can be performed using BDD operations. Since this work is only for real-time systems, all continuous variables can only change with a rate of one. Therefore, this paper extends this work to support continuous variables that can change at any rate within a range in order to allow for the symbolic modeling checking of AMS circuits with BDDs. In preparation for the application of the Boolean model checking method, the LHPN is converted to a Boolean symbolic model and the TCTL is converted into a *timed  $\mu$*  ( $T\mu$ ) property. Finally, model checking is performed and a verification result is obtained.

## II. MOTIVATING EXAMPLE

This paper uses the switched capacitor integrator shown in Fig. 2 as a running example. This circuit takes as input a 5 kHz square wave that varies from  $-1$  V to  $+1$  V and generates a

\*This research is supported by SRC contract 2005-TJ-1357 and an SRC Graduate Fellowship.

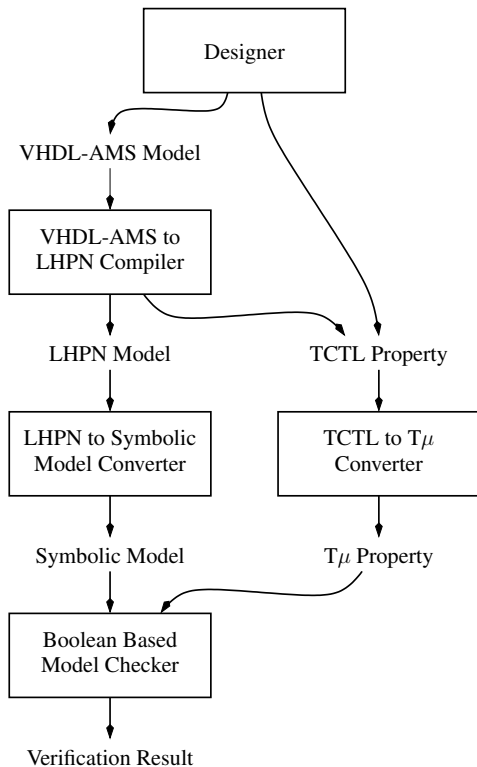


Fig. 1. Verification tool flow.

triangle wave as output representing the integral of the input voltage. Discrete-time integrators typically utilize switched capacitor circuits to accumulate charge which can cause gain errors in the integrator due to capacitor mismatch. Therefore, the output voltage in our model is allowed to have a slew rate anywhere between 18 to 22 mV/ $\mu$ s to represent a  $\pm 10$  percent variance in circuit parameters. The verification goal is to ensure that  $V_{out}$  never saturates (i.e., it is always between  $-2000$  mV and  $2000$  mV). An experienced analog circuit designer may realize the potential of this circuit to fail. However, a very specific and unusual SPICE simulation is required to demonstrate this failure. Specifically, a failure only appears in a simulation where capacitor mismatch results in a different slew rate when charging the capacitor versus when discharging the capacitor. Furthermore, it is highly unlikely that a simulation allowing for random uncertainty in the system variables would reveal the error. Therefore, a formal verification approach is beneficial.

Using a subset of VHDL-AMS, the circuit in Fig. 2 can be modeled as shown in Fig. 3. The VHDL-AMS subset that is supported allows variables of types **std\_logic** for representing Boolean signals and **real** for representing continuous quantities. Continuous variables can be initialized using **break** statements. The rates of continuous variables can be updated using simultaneous statements such as the **if-use** and **case-use** statements. Sequential behavior can be specified using **process** statements without sensitivity lists. Within a **process**, supported statements are **wait**, signal assignment, **if-then**, **case**, and **while-loop**. Finally, **assert** statements can be used to state basic safety properties about the system. For convenience, our VHDL-AMS descriptions also use procedures defined in the

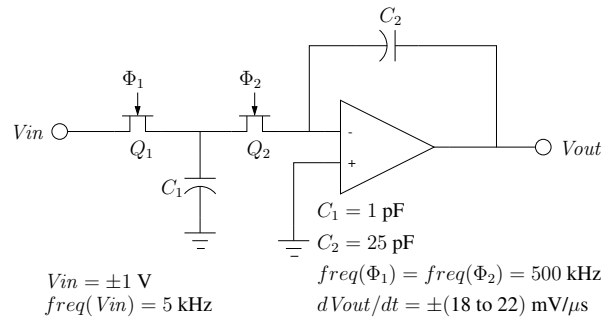


Fig. 2. Circuit diagram for a switched capacitor integrator.

```

library IEEE;
use IEEE.std_logic_1164.all;
use work.handshake.all;
use work.nondeterminism.all;
entity integrator is
end integrator;
architecture switchCap of integrator is
    quantity Vout:real;
    signal Vin:std_logic := '0';
begin
    break Vout => -1000.0; --Initial value
    if Vin='0' use
        Vout'dot == span(18.0, 22.0);
    elsif Vin = '1' use
        Vout'dot == span(-22.0, -18.0);
    end use;
    process begin
        assign(Vin,'1',100,100);
        assign(Vin,'0',100,100);
    end process;
    assert (Vout'above(-2000.0) and
        not Vout'above(2000.0))
        report ``error``
        severity failure;
end switchCap;

```

Fig. 3. VHDL-AMS for a switched capacitor integrator.

**handshake** and **nondeterminism** packages [15]. For example, the **assign** procedure performs an assignment to a signal at some random time within a bounded range specified by its parameters and waits until the assignment has been performed before returning. The **span** procedure takes two real values and returns a random value within that range. The **span** procedure is used to assign a range of rate to a continuous variable.

While similar to the VHDL-AMS description in [13], the VHDL-AMS shown in Fig. 3 is more concise because our analysis allows rates to be specified as ranges. This model tracks the real quantity  $V_{out}$  that represents the output voltage. The Boolean variable  $V_{in}$  determines the rate of  $V_{out}$  using the **if-use** statements. When  $V_{in}$  is 0,  $V_{out}$  increases at a rate between 18 and 22 mV/ $\mu$ s and when  $V_{in}$  is 1,  $V_{out}$  decreases at a rate between  $-22$  and  $-18$  mV/ $\mu$ s. Initially  $V_{out}$  is  $-1000$  mV and increasing between 18 and 22 mV/ $\mu$ s. After 100  $\mu$ s,  $V_{in}$  is assigned to 1 by the **assign** function which causes  $V_{out}$  to begin decreasing at a rate of  $-22$  to  $-18$  mV/ $\mu$ s. The **assert** statement is used to check if  $V_{out}$  falls below  $-2000$  mV or goes above 2000 mV.

### III. LABELED HYBRID PETRI NETS

Our VHDL-AMS description is compiled into an LHPN for analysis. An LHPN is a Petri net model developed to represent AMS circuits with the goal of being easily generated from VHDL-AMS descriptions. The model is inspired by features in both hybrid Petri nets [4] and hybrid automata [1]. An LHPN is a tuple  $N = \langle P, T, B, V, F, L, M_0, S_0, Q_0, R_0 \rangle$ :

- $P$  : is a finite set of places;
- $T$  : is a finite set of transitions;
- $B$  : is a finite set of Boolean signals;
- $V$  : is a finite set of continuous variables;
- $F \subseteq (P \times T) \cup (T \times P)$  is the flow relation;
- $L$  : is a tuple of labels defined below;
- $M_0 \subseteq P$  is the set of initially marked places;
- $S_0$  : is the set of initial Boolean signal values;
- $Q_0$  : is the set of initial continuous variable values;
- $R_0$  : is the set of initial continuous variable rates.

A key component of LHPNs are the labels. Some labels contain *hybrid separation logic* (HSL) formulas which are a Boolean combination of Boolean variables and separation predicates (a restricted form of inequalities relating real variables). HSL is an extension of separation logic (sometimes referred to as difference logic) that allows for non-unit slopes on the separation predicates. These formulas satisfy the following grammar:

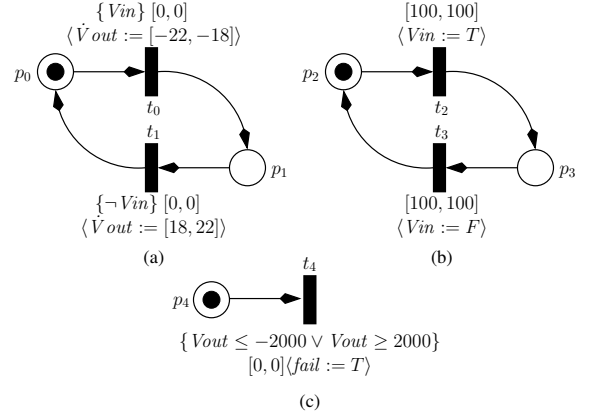
$$\phi ::= \mathbf{true} \mid \mathbf{false} \mid b_i \mid \neg\phi \mid \phi \wedge \phi \mid c_i x_i \geq c_j x_j + c$$

where  $b_i$  are Boolean variables,  $x_i$  and  $x_j$  are continuous variables, and  $c_i$ ,  $c_j$ , and  $c$  are constants from the set of rational numbers,  $\mathbb{Q}$ . Each transition  $t \in T$  is labeled using the functions defined in  $L = \langle E, D, BA, VA, RA \rangle$ :

- $E : T \rightarrow \phi$  labels each  $t$  with an enabling condition;
- $D : T \rightarrow \mathbb{Q} \times (\mathbb{Q} \cup \{\infty\})$  labels each  $t$  with a lower and upper bound delay value,  $[d_l, d_u]$ ;
- $BA : T \rightarrow 2^{(B \times \{0,1\})}$  labels each  $t$  with Boolean signal assignments made when  $t$  fires;
- $VA : T \rightarrow 2^{(V \times \mathbb{Q})}$  labels each  $t$  with continuous variable assignments made when  $t$  fires;
- $RA : T \rightarrow 2^{(V \times \mathbb{Q} \times \mathbb{Q})}$  labels each transition with a range of rate assignments,  $[r_l, r_u]$ , made when  $t$  fires.

The LHPN shown in Fig. 4 is automatically generated from the VHDL-AMS model in Fig. 3. The **break** statement sets the initial value for  $V_{out}$ . The **if-use** statement is compiled into the LHPN in Fig. 4a. The **process** statement is compiled into the LHPN in Fig. 4b. The **assert** statement is compiled into the LHPN shown in Fig. 4c which fires a transition to set the Boolean signal *fail* to true when the assertion is violated.

Formal semantics of LHPNs are given in [13, 18]. Intuitively, transitions in LHPNs are controlled by enabling conditions and timing constraints. When the enabling condition becomes satisfied, the clock on the transition begins, and the



$$Q_0 = \{V_{out} = -1000\} \quad R_0 = \{\dot{V}_{out} = [18, 22]\} \quad S_0 = \{\neg Vin, \neg fail\}$$

Fig. 4. LHPN for the switched capacitor integrator.

transition fires sometime after the clock reaches its lower bound and before it exceeds its upper bound. Upon firing, the discrete marking is updated by removing tokens from the preset places of the transition and placing tokens in the postset places of the transition. Additionally, assignments are made to Boolean signals, continuous variables, and rates of continuous variables. For the LHPN in Fig. 4, the marking is initially  $\{p_0, p_2, p_4\}$ , the Boolean signal  $V_{in}$  is false, the continuous variable  $V_{out}$  is  $-1000$ , and  $V_{out}$  is increasing at a rate of  $18$  to  $22$  mV/ $\mu$ s. After  $100 \mu$ s,  $t_2$  fires resulting in  $p_2$  becoming unmarked,  $p_3$  becoming marked, and the assignment of true to  $V_{in}$ . This assignment causes the enabling and immediate firing of  $t_0$  and thus the assignment of  $-22$  to  $-18$  mV/ $\mu$ s to the rate for  $V_{out}$ .

### IV. BOOLEAN REPRESENTATION

The verification algorithm relies on performing Boolean operations using BDDs. Thus, it is necessary to efficiently represent HSL formulas as Boolean formulas. This requires a canonical representation of HSL separation predicates.<sup>1</sup> The canonical representation is of the form  $c_i x_i \geq c_j x_j + c$  with the following restrictions where  $x_0$  is a special variable representing zero:

- The continuous variables  $x_i$  and  $x_j$  are distinct.
- If  $x_i = x_0$  or  $x_j = x_0$  then the corresponding constant  $c_i$  or  $c_j$  is one.
- The constants  $c_i$  and  $c_j$  are not both negative.
- If  $c_i$  or  $c_j$  is negative (but not both), then in the ordered set of real variables,  $x_i$  comes before  $x_j$ .
- The constants  $c_i$  and  $c_j$  are arbitrarily large integers with a greatest common denominator of one.
- The constant  $c$  is a rational number using arbitrarily large integers as the numerator and the denominator.

Using these restrictions and the fact that separation predicates of the form  $c_i x_i > c_j x_j + c$  are equivalent to  $c_j x_j \geq c_i x_i + -c$ , any separation predicate can be represented in a unique way.

<sup>1</sup>A similar approach is suggested for octagonal polyhedra in [14].

For clarity, we show separation predicates throughout the remainder of this paper, however they are actually mapped to BDD variables in their canonical form.

Using the canonical form of separation predicates, relationships among continuous variables in LHPNs can now be represented in a Boolean manner. It is also necessary to maintain relationships among the timers on transitions. This necessitates the creation of additional continuous variables, denoted by  $c_t$ , to represent the value of the clock on transition  $t$ .

To complete the Boolean representation, BDD variables are created for each place in the LHPN to indicate if the place is marked, for each Boolean signal (denoted using the Boolean signal's name), and for each transition's clock to indicate if the clock is active or inactive (denoted as  $a_t$  for the clock on transition  $t$ ). Finally, a BDD variable is created for each possible rate on each continuous variable. These BDD variables are known as Boolean rate variables and are denoted as  $\dot{v}_{[r_l, r_u]}$  for the BDD variable corresponding to the continuous variable  $\dot{v}$  having rate  $[r_l, r_u]$ . Additionally,  $\bullet t$  and  $t\bullet$  represent the preset and postset of  $t$ , respectively, and the notation  $(t\bullet) := T$  means that each element in the postset of  $t$  is assigned true. The notation  $\overline{t\bullet - \bullet t}$  is used as a condition to ensure that the places in the postset of the transition  $t$ , aside from places that form a self loop, are not marked.

## V. SYMBOLIC MODEL

In order for analysis to proceed, a symbolic model is generated that contains the essential information for analysis. The symbolic model consists of three components: an *invariant*, a set of possible rates, and a set of *guarded commands*.

The invariant ( $\phi_{\mathcal{I}}$ ) is an HSL statement that must be satisfied in every state of the system. First, it states that only the discrete states (represented by  $\Phi$ ) can be reached. The formula  $\Phi$  is found by performing a state space exploration of the LHPN neglecting the continuous variables. The discrete state space exploration is based on the Petri net algorithm described in [16] with extensions to include values of Boolean signals and Boolean rate variables in the state space. In other words,  $\Phi$  is a formula over the Boolean variables for the Petri net marking, Boolean signals, and Boolean rate variables. Next,  $\phi_{\mathcal{I}}$  states that for a transition's clock to be active, the preset must be marked, the enabling condition must be satisfied, and the clock must be greater than zero but not greater than its upper bound. This portion of  $\phi_{\mathcal{I}}$  prevents an active clock from exceeding its upper bound. The last part of  $\phi_{\mathcal{I}}$  states that if a transition's clock is not active it must either have an unmarked place in its preset or the *non-strict inverse* ( $\widetilde{E}(t)$ ) of the enabling condition must be satisfied. In the non-strict inverse, all  $\geq$  separation predicates become  $\leq$  separation predicates and vice-versa. The last two portions of  $\phi_{\mathcal{I}}$  when taken together enforce the activation or deactivation of a clock if a changing continuous variable should cause an enabling condition to change evaluation. The invariant is defined formally as follows:

$$\phi_{\mathcal{I}} = \Phi \wedge \bigwedge_{t \in T} (a_t \Rightarrow \bullet t \wedge E(t) \wedge 0 \leq c_t \leq u(t)) \wedge (\overline{a_t} \Rightarrow \overline{\bullet t} \vee \widetilde{E}(t))$$

For the integrator example in Fig. 4, the invariant is:

$$\begin{aligned} \phi_{\mathcal{I}} = & ((p_0 \overline{p_1} p_2 \overline{p_3} \overline{Vin} \overline{\dot{V}out}_{[-22, -18]} \dot{V}out_{[18, 22]}) \vee \\ & (p_0 \overline{p_1} \overline{p_2} p_3 \overline{Vin} \overline{\dot{V}out}_{[-22, -18]} \dot{V}out_{[18, 22]}) \vee \\ & (\overline{p_0} p_1 \overline{p_2} p_3 \overline{Vin} \dot{V}out_{[-22, -18]} \overline{\dot{V}out}_{[18, 22]}) \vee \\ & (\overline{p_0} p_1 p_2 \overline{p_3} \overline{Vin} \dot{V}out_{[-22, -18]} \overline{\dot{V}out}_{[18, 22]}) \wedge \\ & (a_{t_0} \Rightarrow p_0 \wedge Vin \wedge c_{t_0} = 0) \wedge (\overline{a_{t_0}} \Rightarrow \overline{p_0} \vee \overline{Vin}) \wedge \\ & (a_{t_1} \Rightarrow p_1 \wedge \overline{Vin} \wedge c_{t_1} = 0) \wedge (\overline{a_{t_1}} \Rightarrow \overline{p_1} \vee Vin) \wedge \\ & (a_{t_2} \Rightarrow p_2 \wedge 0 \leq c_{t_2} \leq 100) \wedge (\overline{a_{t_2}} \Rightarrow \overline{p_2}) \wedge \\ & (a_{t_3} \Rightarrow p_3 \wedge 0 \leq c_{t_3} \leq 100) \wedge (\overline{a_{t_3}} \Rightarrow \overline{p_3}) \wedge \\ & (a_{t_4} \Rightarrow p_4 \wedge c_{t_4} = 0 \wedge \\ & (Vout \leq -2000 \vee Vout \geq 2000)) \wedge \\ & (\overline{a_{t_4}} \Rightarrow \overline{p_4} \vee (Vout \geq -2000 \wedge Vout \leq 2000)) \end{aligned}$$

The set of possible rates ( $\mathcal{R}$ ) consist of an HSL statement indicating a possible Boolean rate assignment and the set of rate assignments to continuous variables corresponding to the statement ( $\langle \phi_R, R \rangle$ ). This set is constructed from  $\Phi$ , the Boolean state set, by existentially abstracting all non-rate Boolean variables. Each product term corresponds to a  $\phi_R$  of a pair in  $\mathcal{R}$ . The Boolean rate assignment sets ( $R$ ) are built from the product terms. For example, the possible rate set for Fig. 4 is:

$$\begin{aligned} \mathcal{R} = & \{ \langle \overline{\dot{V}out}_{[-22, -18]} \wedge \overline{\dot{V}out}_{[18, 22]}, \\ & \{ \dot{V}out := [-22, -18] \} \rangle, \\ & \langle \overline{\dot{V}out}_{[-22, -18]} \wedge \dot{V}out_{[18, 22]}, \\ & \{ \dot{V}out := [18, 22] \} \rangle \end{aligned}$$

The set of guarded commands ( $\mathcal{C}$ ) is used to determine in each state which transitions are enabled and the effect on the state due to the firing of a transition. It is constructed using a set of *primary guarded commands* ( $\mathcal{C}_P$ ) and a set of *secondary guarded commands* ( $\mathcal{C}_S$ ). Each guarded command consists of a *guard*,  $\phi_G$ , represented using an HSL formula and a set of *commands*,  $\mathcal{A}$ , to be performed when the guard is satisfied.

A primary guarded command is created for each transition  $t \in T$ . The guard for transition  $t$  ensures that the preset for  $t$  is marked, the enabling condition on  $t$  is satisfied, and the clock associated with  $t$  is active and exceeds its lower bound. The commands for transition  $t$  cause the postset of  $t$  to become marked and the application of the assignments associated with  $t$ . Formally, the set of primary guarded commands is defined as follows:

$$\mathcal{C}_P = \bigcup_{t \in T} \{ \langle \phi_{G_P}(t), \mathcal{A}_P(t) \rangle \}$$

where  $\phi_{G_P}(t) = (\bullet t \wedge \overline{t\bullet - \bullet t} \wedge E(t) \wedge a_t \wedge c_t \geq l(t))$  and  $\mathcal{A}_P(t) = \{ (\bullet t - t\bullet) := F, (t\bullet) := T, a_t := F, c_t := [-\infty, \infty], BA(t), VA(t), RA(t) \}$ . The primary guarded command for transition  $t_2$  in Fig. 4 is:

$$\begin{aligned} \phi_{G_P}(t_2) &= p_2 \wedge \overline{p_3} \wedge a_{t_2} \wedge c_{t_2} \geq 100 \\ \mathcal{A}_P(t_2) &= \{ p_2 := F, p_3 := T, Vin := T, \\ & a_{t_2} := F, c_{t_2} := [-\infty, \infty] \} \end{aligned}$$

Two secondary guarded commands are created for each transition  $t \in T$ , one to activate and one to deactivate the clock associated with  $t$ . The first one activates the clock for  $t$  and sets it to zero when its preset is marked and its enabling condition is true. The second one deactivates the clock when  $t$  is no longer enabled and sets its values to  $[-\infty, \infty]$ . This has the effect of removing the clock from the state space. The set of secondary guarded commands is defined as follows:

$$\mathcal{C}_S = \bigcup_{t \in T} \{ \langle \phi_{G_{SA}}(t), \mathcal{A}_{SA}(t) \rangle, \langle \phi_{G_{SD}}(t), \mathcal{A}_{SD}(t) \rangle \}$$

where  $\phi_{G_{SA}}(t) = \bullet t \wedge E(t) \wedge \overline{a_t}$ ,  $\mathcal{A}_{SA}(t) = \{a_t := T, c_t := [0, 0]\}$ ,  $\phi_{G_{SD}}(t) = (\overline{\bullet t} \vee \overline{E}(t)) \wedge a_t$ , and  $\mathcal{A}_{SD}(t) = \{a_t := F, c_t := [-\infty, \infty]\}$ . The activating and deactivating guarded commands for transition  $t_0$  in Fig. 4 are:

$$\begin{aligned} \phi_{G_{SA}}(t_0) &= p_0 \wedge \overline{Vin} \wedge \overline{a_{t_0}} \\ \mathcal{A}_{SA}(t_0) &= \{a_{t_0} := T, c_{t_0} := [0, 0]\} \\ \phi_{G_{SD}}(t_0) &= (\overline{p_0} \vee \overline{Vin}) \wedge a_{t_0} \\ \mathcal{A}_{SD}(t_0) &= \{a_{t_0} := F, c_{t_0} := [-\infty, \infty]\} \end{aligned}$$

The sets  $\mathcal{C}_P$  and  $\mathcal{C}_S$  are merged to form the set  $\mathcal{C}$ . It is necessary to merge these commands because the firing of a transition may result in the activation or deactivation of clocks associated with other transitions by changing the marking or the values of the Boolean or continuous variables. Only the intuition behind the merging process is described here. The basic idea is that for each transition,  $t$ , the effect of its assignments associated with its primary guarded command  $\mathcal{A}_P(t)$  must be checked against the guards  $\phi_{G_{SA}}(t')$  and  $\phi_{G_{SD}}(t')$  for each other transition  $t'$  to determine if the assignment may have enabled the guard. If the assignments have no effect on the guard or disable it, then the secondary for  $t'$  is not merged with the primary for  $t$ . If the assignment would make the guard true, then the commands associated with the secondary must be combined with those for the primary. Finally, if the assignment may have changed the guard's evaluation, then two guarded commands must be constructed. One is for the case in which the guard for the secondary is true in which the commands are merged, and the other is for when the guard is false in which the secondary commands are not merged. For example, since the primary guarded command for  $t_2$  assigns  $Vin$  to true, a condition in the guard of the activating guarded command on  $t_0$ , they are merged into the guarded command shown below:

$$\begin{aligned} \phi_G(t_2, t_0) &= p_0 \wedge p_2 \wedge \overline{p_3} \wedge \overline{a_{t_0}} \wedge a_{t_2} \wedge c_{t_2} \geq 100 \\ \mathcal{A}(t_2, t_0) &= \{p_2 := F, p_3 := T, Vin := T, \\ &\quad a_{t_0} := T, c_{t_0} := [0, 0], \\ &\quad a_{t_2} := F, c_{t_2} := [-\infty, \infty]\} \end{aligned}$$

## VI. SPECIFYING PROPERTIES

Properties to be checked are specified using a dense real-time version of CTL known as TCTL. For example, the TCTL property to check for the integrator is  $AG(\neg fail)$ . This property is automatically generated from the **assert** statement in the VHDL-AMS code. More complex properties can be manually provided by the user, if desired.

A TCTL property is translated into a  $\mathbf{T}\mu$  calculus formula as described in [9].  $\mathbf{T}\mu$  calculus has the following grammar as defined in [9]:

$$\varphi ::= Y \mid \phi \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \triangleright \varphi_2 \mid z.\varphi \mid \mu Y.\varphi \mid \nu Y.\varphi$$

where  $\phi$  is an HSL formula,  $z$  is a *specification clock variable*, and  $Y$  is a *formula variable* used in fixpoint computation. The next operator “ $\triangleright$ ” means that  $\varphi_1$  is true as time elapses until a discrete transition is taken resulting in  $\varphi_2$ . When the specification clock variable  $z$  is assigned to zero in  $\varphi$ ,  $z.\varphi$  is true. The expressions  $\mu Y.\varphi$  and  $\nu Y.\varphi$  are the least fixpoint and greatest fixpoint, respectively, of  $\varphi$  with the formula variable  $Y$  bound inside  $\varphi$ . The property for the integrator gets transformed into the following  $\mathbf{T}\mu$  formula:

$$\phi_{init} \implies \neg \mu Y.[fail \vee (\mathbf{true} \triangleright Y)]$$

where  $\phi_{init}$  is the initial set of states:

$$\begin{aligned} \phi_{init} &= p_0 \overline{p_1} p_2 \overline{p_3} p_4 \overline{Vin} \overline{Fail} \overline{Vout}_{[-22, -18]} \dot{Vout}_{[18, 22]} \\ &\quad \overline{a_{t_0}} \overline{a_{t_1}} a_{t_2} \overline{a_{t_3}} \overline{a_{t_4}} \wedge c_{t_2} = 0 \wedge Vout = -1000 \end{aligned}$$

If a state in which *fail* is true cannot be reached from the initial state then the formula evaluates to true.

## VII. SYMBOLIC MODEL CHECKING WITH BDDs

Henzinger et al. describe a symbolic model checking algorithm for *timed automata* in which all continuous variables change at rate one [9]. Seshia and Bryant adapted this algorithm for implementation using BDDs [17]. This adapted algorithm is shown in Fig. 5. It proceeds over the structure of  $\varphi$ , a  $\mathbf{T}\mu$  property, given the symbolic model for the system to be verified. Upon termination of the algorithm, the resulting HSL formula is equivalent to  $\phi_{\mathcal{I}}$  if the property is satisfied and the model is non-zeno. The outline of the algorithm in Fig. 5 is applicable to models with rates other than one [2], but extensions are necessary to three critical parts of this algorithm. These parts, *assignment* ( $\phi[A]$ ), *weakest precondition* ( $pre(\phi)$ ), and *time elapse* ( $\phi_1 \rightsquigarrow \phi_2$ ), are described below.

When an assignment,  $\phi[A]$ , operation is performed, a set of assignments, as specified by  $A$ , are simultaneously performed. The set,  $A$ , contains assignments to Boolean signals and/or assignments to continuous variables as defined by a guarded command. Assignments to Boolean signals are of the form  $b := T$  or  $b := F$  and are performed on  $\phi$  by calculating the cofactor of  $\phi$  with respect to the positive or negative form of  $b$ , respectively. The assignment set,  $A$ , may also contain assignments to real variables of the form  $x_i := [-\infty, \infty]$  or  $x_i := a$ . When an  $x_i := [-\infty, \infty]$  assignment is made, all BDD variables in  $\phi$  mapping to separation predicates containing  $x_i$  are existentially quantified. When an assignment of the form  $x_i := a$  is made, all BDD variables in  $\phi$  containing  $x_i$  are found and BDD substitutions to  $\phi$  are performed using the BDD composition operation such that  $c_i x_i \geq c_j x_j + c \leftarrow x_0 \geq c_j x_j + (c - c_i a)$  and  $c_j x_j \geq c_i x_i + c \leftarrow c_j x_j \geq x_0 + (c + c_i a)$ .

The weakest precondition operation,  $pre(\phi)$ , calculates all the possible states that could have resulted in  $\phi$  by firing discrete transitions. In particular, for each guarded command,

$|\phi| := \phi_{\mathcal{I}} \wedge \phi$   
 $|\neg\phi| := \phi_{\mathcal{I}} \wedge \neg|\phi|$   
 $|\varphi_1 \vee \varphi_2| := |\varphi_1| \vee |\varphi_2|$   
 $|\varphi_1 \triangleright \varphi_2| := |(|\varphi_1| \vee |\varphi_2|) \rightsquigarrow pre(|\varphi_2|)|$   
 $|z.\varphi| := |\varphi|[z := 0]$   
 $|\mu Y.\varphi| :=$  the result of the following iteration:  
 $\phi_{new} := \mathbf{false}$   
**repeat**  
 $\phi_{old} := \phi_{new}$   
 $\phi_{new} := |\varphi[Y := \phi_{old}]|$   
**until**  $(\phi_{new} \implies \phi_{old})$   
**return**  $\phi_{old}$

Fig. 5. Symbolic analysis algorithm (courtesy of [17]).

$\langle \phi_G, \mathcal{A} \rangle \in \mathcal{C}$ , it first performs the assignments ( $\mathcal{A}$ ) to the current set of states, and then applies the guard ( $\phi_G$ ). By taking the disjunction of the result for each guarded command, all possible previous states are determined. Finally  $\phi$  is disjunctively combined with the result, and  $\phi_{\mathcal{I}}$  is conjunctively combined to ensure that impossible states are not introduced into the calculation. This is defined formally below:

$$pre(\phi) \doteq \phi_{\mathcal{I}} \wedge (\phi \vee \bigvee_{\langle \phi_G, \mathcal{A} \rangle \in \mathcal{C}} \phi_G \wedge (\phi_{\mathcal{I}} \wedge \phi)[\mathcal{A}]))$$

The time elapse operation ( $\rightsquigarrow$ ) calculates all the states that can reach  $\phi_2$  by allowing time to elapse while remaining in  $\phi_1$  in between. The general idea of time elapse is that the state region  $\phi_2$  is expanded to include all states that can reach  $\phi_2$  by moving time backward. The result is then intersected with all the states that can result in  $\phi_1$  by moving time backward up to the point where  $\phi_1 \wedge \phi_2$  is no longer satisfied. Fig. 6 presents a visual representation of the time elapse operation. Given an initial state region,  $\phi_2$ , the result of time elapse encompasses  $\phi_2$  plus the region within the dotted lines where  $\phi_1$  is satisfied. The time elapse calculation is performed by iterating over the possible rate set,  $\mathcal{R}$ , and operating on the portion of the state space for which  $\phi_R$  is true. The separation predicates in that portion of the state space are evolved backwards based on the rates for each continuous variable in  $R$ . During this calculation, separation predicates that cannot be guaranteed to remain true are existentially abstracted, and new separation predicates that represent the result of time evolution are introduced.

An example of applying a time elapse and transition precondition step to the integrator example is shown in Fig. 7. Beginning with the state shown in Fig. 7c, applying the time elapse operation in a backwards fashion results in the state shown in Fig. 7b. Similarly, applying the weakest precondition operation to the state in Fig. 7b results in the state shown in Fig. 7a.

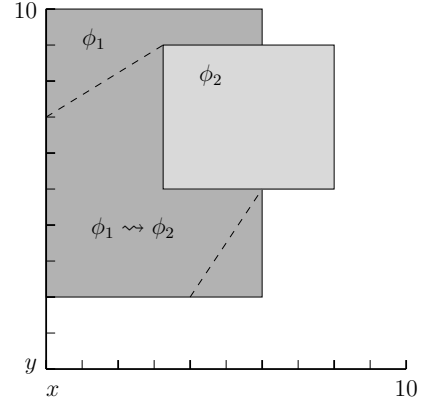


Fig. 6. Visual representation of  $\phi_1 \rightsquigarrow \phi_2$  where  $1 \leq \dot{x} \leq 2$  and  $1 \leq \dot{y} \leq 2$ .

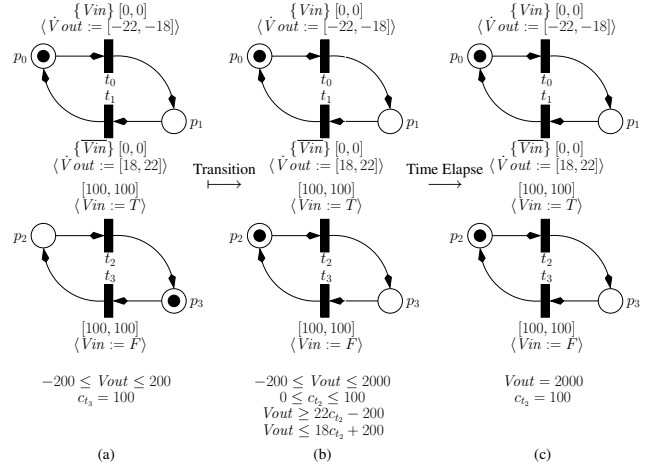


Fig. 7. Application of time elapse and weakest precondition operators to the integrator example. Note that the analysis is backwards beginning from the state shown in (c).

## VIII. CONSTRAINT GENERATION

As new separation predicates are generated and mapped to Boolean variables, *constraints* are added to create relationships with existing Boolean variables and prevent Boolean assignments from being made that are impossible. The use of multiple variable rates in LHPNs, requires an enhanced constraint generation approach over [17] with consideration given to the canonical representation. The constraints are generally added to the HSL formula before a particular variable is going to be existentially abstracted. When constructing transitivity constraints, the signs of the coefficients must be taken into account and both the regular and inverted forms of the variables must be considered. There are two main types of constraints. In the first type, one separation predicate implies the second as follows:

1.  $c_1 x_i \geq c_j x_j + c_1 \Rightarrow c_i x_i \geq c_j x_j + c_2$  if  $c_1 > c_2$ .
2.  $c_j x_j > c_i x_i + -c_1 \Rightarrow c_j x_j > c_i x_i + -c_2$  if  $-c_1 > -c_2$ .
3.  $c_i x_i \geq c_j x_j + c_1 \Rightarrow c_i x_i > c_j x_j + -c_2$  if  $c_1 > -c_2$ .
4.  $c_j x_j > c_i x_i + -c_1 \Rightarrow c_i x_i \geq c_j x_j + c_2$  if  $-c_1 \geq c_2$ .

The second type of constraint is a transitivity constraint—two constraints together imply a third newly created constraint thus forming new relationships between real variables. For example, the two separation predicates  $2x_1 \geq 3x_2 + 2$  and  $5x_2 \geq x_3 + 3$  form the new separation predicate  $10x_1 \geq 3x_3 + 19$  by transitivity and thus a constraint is formed. Given the separation predicates  $c_i x_i \geq c_j x_j + c_1$  and  $c_k x_k \geq c_m x_m + c_2$  (referred to as  $e_1$  and  $e_2$ , respectively), transitivity constraints are formed as follows:

1. If  $x_j = x_k$ , let  $e_3$  represent the separation predicate  $\frac{c_i}{c_j} x_i \geq \frac{c_m}{c_k} x_m + (\frac{c_1}{c_j} + \frac{c_2}{c_k})$  and  $e_4$  represent the separation predicate  $\frac{c_m}{c_k} x_m \geq \frac{c_i}{c_j} x_i + (\frac{-c_2}{c_k} + \frac{-c_1}{c_j})$ .
  - (a) If  $c_j > 0$  and  $c_k > 0$  then  $e_1 \wedge e_2 \Rightarrow e_3$ .
  - (b) If  $c_j < 0$  and  $c_k < 0$  then  $\neg e_1 \wedge \neg e_2 \Rightarrow \neg e_4$ .
  - (c) If  $c_j > 0$  and  $c_k < 0$  then  $\neg e_1 \wedge e_2 \Rightarrow e_4$  and  $\neg e_1 \wedge e_2 \Rightarrow \neg e_3$ .
  - (d) If  $c_j < 0$  and  $c_k > 0$  then  $\neg e_1 \wedge e_2 \Rightarrow e_3$  and  $\neg e_1 \wedge e_2 \Rightarrow \neg e_4$ .
2. If  $x_i = x_k$ , let  $e_3$  represent the separation predicate  $\frac{c_m}{c_k} x_m \geq \frac{c_j}{c_i} x_j + (\frac{-c_2}{c_k} + \frac{c_1}{c_i})$  and  $e_4$  represent the separation predicate  $\frac{c_j}{c_i} x_j \geq \frac{c_m}{c_k} x_m + (\frac{-c_1}{c_i} + \frac{c_2}{c_k})$ .
  - (a) If  $c_i > 0$  and  $c_k > 0$  then  $e_1 \wedge e_2 \Rightarrow e_3$ .
  - (b) If  $c_i < 0$  and  $c_k < 0$  then  $\neg e_1 \wedge \neg e_2 \Rightarrow \neg e_4$ .
  - (c) If  $c_i > 0$  and  $c_k < 0$  then  $\neg e_1 \wedge e_2 \Rightarrow e_4$  and  $\neg e_1 \wedge e_2 \Rightarrow \neg e_3$ .
  - (d) If  $c_i < 0$  and  $c_k > 0$  then  $\neg e_1 \wedge e_2 \Rightarrow e_3$  and  $\neg e_1 \wedge e_2 \Rightarrow \neg e_4$ .

## IX. RESULTS

The VHDL-AMS to LHPN compiler and symbolic modeling checking algorithm described in this paper have been implemented and preliminary results are promising. In addition to checking our tool on several small hybrid system benchmarks, we have also verified various versions of the switched capacitor integrator circuit. In particular, we have experimented with different ranges of rates for  $V_{out}$ . When the lower and upper bound for these rates are equal, our tool determines a few second of CPU time that the property is satisfied (i.e., the circuit does not saturate).<sup>2</sup> When the lower and upper bounds are not equal, our tool determines correctly in a few seconds that the circuit violates the property. This error occurs if the rising slew rate of  $V_{out}$  is consistently larger than the falling slew rate, then charge can build up leading to  $V_{out}$  eventually saturating at the high supply rail.

Saturation of the integrator can be prevented using the circuit shown in Fig. 8. In this circuit, a resistor in the form of a switched capacitor is inserted in parallel with the feedback capacitor. This causes  $V_{out}$  to drift back to 0 V. In other words, if  $V_{out}$  is increasing, it increases faster when it is far below 0 V than when it is near or above 0 V. Therefore, the model for this circuit uses a  $V_{out}$  range of 22 to 24 mV/ $\mu$ s when it is below  $-1000$  mV, and it uses a range of 16 to 22 mV/ $\mu$ s when it is above  $-1000$  mV. A similar modification is made for the ranges of rates when  $V_{out}$  is decreasing. With these changes, verification finds that the circuit no longer saturates.

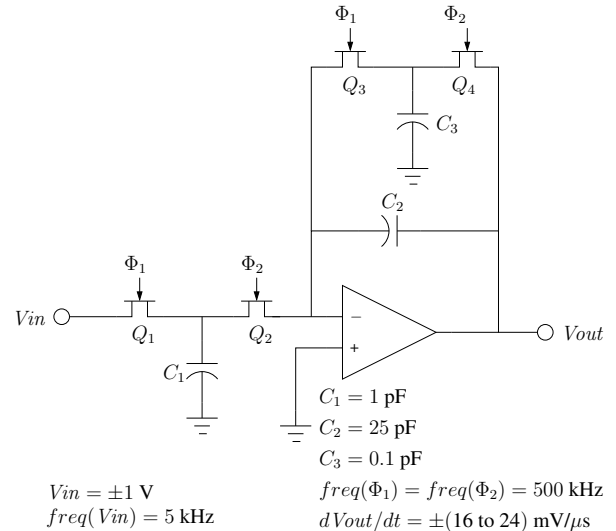


Fig. 8. Circuit diagram of a switched capacitor integrator that has been modified to prevent saturation.

The symbolic model checker described in this paper is the first to support the verification of VHDL-AMS models with ranges of rates. This is accomplished due to the fact that the model checker is designed to work with LHPN models which are developed with the goal of being easily generated from VHDL-AMS descriptions. For comparison purposes, it is possible to hand translate the LHPN model for the switched capacitor integrator into a hybrid automaton. While this is not too complicated for this small LHPN, this translation is in general difficult and can result in a blowup in the size of the hybrid automaton. This hand generated hybrid automaton can then be verified using the hybrid automata tools HyTech [10] and PHAVer [5]. While both tools rapidly verified the integrator example with equal bounds on the rates of change for  $V_{out}$ , they are unable to complete for the integrator with ranges of rates for  $V_{out}$ . Upon closer examination, we determined that this is due to the fact that the state space is unbounded in these cases. To address this problem, we add *by hand* invariants to the hybrid automaton to bound the state space that is explored. Both tools can then complete the verification of the integrator examples in runtimes that are comparable with our tool.

## X. CONCLUSION

To gain acceptance of formal verification by AMS designers, it is crucial to allow them to describe circuits using a method that they are comfortable with. To this end, this paper describes a method for symbolic model checking of AMS circuits described using a subset of VHDL-AMS. These VHDL-AMS descriptions are compiled into LHPNs which are then analyzed using an algorithm that maps separation predicates to Boolean variables allowing for the use of BDDs. This algorithm extends previous work by providing a canonical representation of separation predicates containing two real variables with arbitrary slopes and introduces an expanded constraint generation method. The time elapse calculation, in particular, must also

<sup>2</sup>All tests performed on a 3GHz PentiumIV with 1GB of RAM.

be modified substantially. These extensions are necessary for a BDD based implementation of an algorithm that allows for real variables to change at ranges of rates.

We are currently developing a SPICE-deck front-end which will further improve the ability of AMS designers to use our tool. We are also planning to develop abstraction methods to reduce the number of BDD variables that are created and to help alleviate the state explosion problem that can occur as we apply these methods to larger scale examples. Additionally, we believe that many of the methods described in this paper may lend themselves well to a bounded model checking approach using SAT and/or SMT systems. In this approach, the transition relation and time elapse calculation would introduce next state variables and time step variables at each step of the analysis. Finally, we plan to investigate methods to improve user feedback when a failure is detected.

#### ACKNOWLEDGEMENTS

We would like to thank Goran Frehse of VERIMAG for his help with PHAVER. We would also like to thank Sanjit Seshia of UC Berkeley, Randal Bryant of CMU, Reid Harrison of the University of Utah, Robert Kurshan of Cadence, and Kevin Jones of Rambus for their comments on this work.

#### REFERENCES

- [1] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.
- [2] R. Alur, T. A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. In *IEEE Transactions on Software Engineering*, pages 181–201, 1996.
- [3] T. Dang, A. Donze, and O. Maler. Verification of analog and mixed-signal circuits using hybrid systems techniques. In *Formal Methods for Computer Aided Design*, 2004.
- [4] R. David and H. Alla. On hybrid petri nets. *Discrete Event Dynamic Systems: Theory and Applications*, 11:9–40, Jan. 2001.
- [5] G. Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In Manfred Morari and Lothar Thiele, editors, *HSCC*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2005.
- [6] G. Frehse, B. H. Krogh, and R. A. Rutenbar. Verifying analog oscillator circuits using forward/backward refinement. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 257–262. IEEE Computer Society Press, 2006.
- [7] S. Gupta, B. H. Krogh, and R. A. Rutenbar. Towards formal verification of analog designs. In *International Conference on Computer-Aided Design*, pages 210–217, 2004.
- [8] W. Hartong, L. Hedrich, and E. Barke. Model checking algorithms for analog verification. In *Design Automation Conference*, pages 542–547, 2002.
- [9] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Society Press.
- [10] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.
- [11] IBS Corporation. Industry reports, 2003.
- [12] R. P. Kurshan and K. L. McMillan. Analysis of digital circuits through symbolic reduction. *IEEE Transactions on Computer-Aided Design*, 10(11):1356–1371, November 1991.
- [13] S. Little, N. Seegmiller, D. Walter, and C. J. Myers. Verification of analog/mixed-signal circuits using labeled hybrid petri nets. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, November 2006.
- [14] A. Miné. The octagon abstract domain. In *Analyzing, Slicing, and Transformation*, pages 310–319. IEEE Computer Society Press, October 2001.
- [15] C. Myers. *Asynchronous Circuit Design*. Wiley, 2001.
- [16] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In R. Valette, editor, *Proc. of the 15th Int. Conf. on Application and Theory of Petri Nets (PNPM'94), Zaragoza, Spain*, LNCS 815, pages 416–435. Springer, June 1994.
- [17] S. A. Seshia and R. E. Bryant. Unbounded, fully symbolic model checking of timed automata using boolean methods. In *Proc. International Workshop on Computer Aided Verification*, pages 154–166, 2003.
- [18] D. Walter. *Verification of Analog and Mixed-Signal Circuits Using Binary Decision Diagrams and Predicate Abstraction*. PhD thesis, University of Utah, 2007.