Bounded Model Checking of Analog and Mixed-Signal Circuits Using an SMT Solver*

David Walter, Scott Little, and Chris Myers

University of Utah, Salt Lake City, UT 84112, USA
{dwalter,little,myers}@vlsigroup.ece.utah.edu

Abstract. This paper presents a bounded model checking algorithm for the verification of *analog and mixed-signal* (AMS) circuits using a *satisfiability modulo theories* (SMT) solver. The systems are modeled in *VHDL-AMS*, a hardware description language for AMS circuits. In this model, system safety properties are specified as assertion statements. 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. The verification method begins by transforming the LHPN model into an SMT formula composed of the initial state, the transition relation unrolled for a specified number of iterations, and the complement of the assertion in each set of state variables. When this formula evaluates to true, this indicates a violation of the assertion and an error trace is reported. This method has been implemented and preliminary results are promising.

1 Introduction

To date, there has been relatively little research in the formal verification of *analog and mixed-signal* (AMS) circuits. Perhaps the first work in this area is from Kurshan and McMillan in which analog circuits are represented as finite state models [1]. Hartong et al. verify analog circuits by dividing the continuous state space into regions that are represented in a Boolean manner [2]. This allows them to use Boolean-based verification but with significant loss in accuracy. Hybrid system tools have also been adapted to verify AMS circuits. Gupta et al. utilize `CheckMate` to verify a tunnel diode oscillator and a delta-sigma modulator [3]. In [4], Dang et al. use d/dt to verify a biquad low-pass filter. In [5], Frehse et al. use `PHAVer` to verify analog oscillator circuits. These tools are very accurate but also very computationally complex. These approaches also require a user to describe an AMS circuit using a *hybrid automaton* which is unfamiliar to most AMS designers. In [6], Little et al. use *difference bound matrices* (DBMs) to verify AMS circuits. This method, however, only supports constant rates of change for continuous variables and conservatively abstracts the continuous state space. In [7], Walter et al. present a BDD model checking algorithm for verifying AMS circuits. This method, however, can have substantial memory requirements.

* Support from SRC contract 2005-TJ-1357 and an SRC Graduate Fellowship.

The goal of this paper is to develop a method for verifying AMS circuits using a *satisfiability modulo theories* (SMT) solver. The SMT problem is a generalization of the *Boolean Satisfiability* (SAT) problem where Boolean variables are replaced by predicates from various background theories [8]. These theories may include linear arithmetic over reals and integers, uninterpreted functions, and the theories of various data structures such as lists, arrays, and bit vectors [9,10,11,12,13,14,15]. Initial SMT solver implementations functioned by translating SMT instances into SAT instances and passing those SAT instances to a SAT solver. For example, to support integer arithmetic, multiple Boolean variables are used as a bit representation for integers and the necessary integer theories are specified as Boolean operations on those individual bit variables. This can result in extremely large SAT instances; however, existing SAT solvers can be used directly without modification. Therefore, as SAT solvers improve, so do the SMT solvers. This approach, however, can be severely restricting. The loss of higher level knowledge of the underlying theories requires the SAT solver to work harder to discover simple concepts [16]. This problem is made even more difficult by the large SAT instances that result.

More recent SMT solvers [17,15,18] closely integrate theory-specific solvers with a DPLL (Davis-Putnam-Logemann-Loveland) approach to Boolean satisfiability [8]. These types of SMT solvers are often referred to as DPLL(T) [15]. In this type of architecture, the DPLL-based SAT solver passes conjunctions of predicates belonging to theory T to a specialized solver. The specialized solver is then responsible for deciding feasibility of those predicates. Additionally, the particular theory solver must be able to explain the reasons for infeasibility. Recent work applies SMT solvers to the bounded model checking of software [16]. There are a number of SMT solvers including `Barcelogic` [15,18], `MathSAT` [17], and `Yices` [19]. The `Barcelogic` solver supports difference logic over integers and equality with uninterpreted functions. The `MathSAT` solver currently supports theories of equality, uninterpreted functions, separation logic, and linear arithmetic over reals and integers. `Yices` includes an incremental Simplex algorithm for the theory of linear arithmetic that is tightly integrated within the DPLL framework. `Yices` strong ability to work with the theory of linear arithmetic makes it particularly well suited for hybrid system model checking. For this reason, `Yices` is selected as the SMT solver for the bounded model checker described in this paper.

This paper describes a bounded model checking algorithm for the verification of AMS circuits. The model checker begins with a VHDL-AMS description of an AMS circuit that is compiled into a *labeled hybrid Petri net* (LHPN). Next, the LHPN model is converted into an SMT formula which includes a set of Boolean and continuous state variables for each of the specified number of iterations. The formula is composed of the initial state, the transition relation, and the negation of the assertion statement. The SMT solver `Yices` is used to evaluate this formula. When a satisfying assignment is found, this indicates a failure, and an error trace is reported. This method has been implemented and preliminary results are promising.

2 Motivating Example

The switched capacitor integrator shown in Figure 1 is used as a running example throughout this paper. This circuit takes as input a 5 *kHz* square wave that varies from −1000 mV to 1000 mV and generates a 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/μs to represent a ±10 percent variance in circuit parameters. The verification goal is to ensure that *Vout* 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 SPICE simulation is required to demonstrate this failure where the output voltage always increases at a faster rate than it decreases. Furthermore, it is highly unlikely that a simulation allowing for random uncertainty in the system variables would reveal the error [20]. Therefore, a formal verification approach is beneficial.

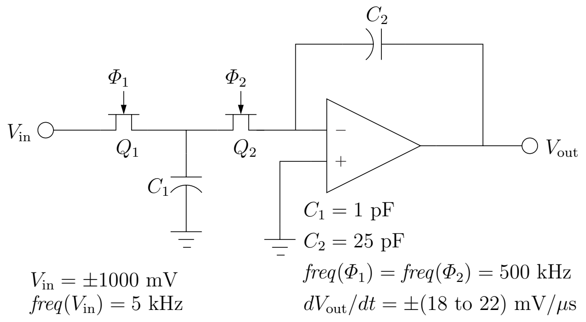

C_2
Φ_1
Φ_2
V_{in}
Q_1
Q_2
C_1
−
+
V_{out}
$C_1 = 1$ pF
$C_2 = 25$ pF
$freq(\Phi_1) = freq(\Phi_2) = 500$ kHz
$dV_{\text{out}}/dt = \pm(18 \text{ to } 22)$ mV/μs
$V_{\text{in}} = \pm 1000$ mV
$freq(V_{\text{in}}) = 5$ kHz

Fig. 1. Circuit diagram of a switched capacitor integrator

VHDL-AMS is a hardware description language that includes extensions specifically for describing analog and mixed-signal circuits. VHDL-AMS was designed to allow a textual description of AMS circuits which can be simulated. By providing a VHDL-AMS front-end to our tool, many of the hurdles associated with verification can potentially be avoided because designers who are already familiar with VHDL-AMS are not required to learn abstract modeling methods. Our VHDL-AMS compiler is built using methods described in [21] and currently works with a subset of the VHDL-AMS language. Methods for generating LHPNs from many VHDL statements for representing digital systems are described in [21]. Specifically, variables of types **std_logic** for representing Boolean signals are allowed and sequential behavior can be specified using **process** statements without sensitivity lists. Within a **process**, supported statements are **wait**, signal assignment, **if-use**, **case**, and **while-loop**.

Simulators that support the AMS extensions to VHDL seem to vary in the semantics that are implemented. Therefore, a subset of the AMS extensions have been selected such that the semantics seem to be fairly consistent across simulators. The supported subset of VHDL-AMS allows the creation of a continuous value using a **quantity** of type **real**, the initialization of continuous variables using **break** statements, and the assignments of rates to real quantities using the **'dot** notation within simultaneous **if-use** and **case-use** statements. Additionally, the use of **'above** to test the value of real quantities, and the specification of properties using **assert** statements is allowed. For convenience, VHDL-AMS descriptions also use procedures defined in the `handshake` and `nondeterminism` packages [22]. 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 rates to a continuous variable. Figure 2 shows a VHDL-AMS description for the circuit in Figure 1. The **break** statement sets the initial value for *Vout*. The **if-use** statement determines the rate of *Vout*. When *Vin* is **false**, *Vout* increases at a rate between 18 and 22 mV/μs. When *Vin* is **true**, it decreases at a rate between -22 and -18 mV/μs. The **process** statement controls *Vin*. Finally, an **assert** statement checks if *Vout* saturates.

```
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. 2. VHDL-AMS for a switched capacitor integrator

3 Labeled Hybrid Petri Nets

Our VHDL-AMS descriptions are compiled automatically into LHPN models. LHPNs were developed specifically to model AMS circuits. The model is inspired by features in both hybrid Petri nets [23] and hybrid automata [24]. While LHPNs are only described briefly here, a complete definition with formal semantics can be found in [25]. An LHPN is defined as a directed graph with labels on places and transitions. 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 ranges of values for each continuous variable and;
- R_0 : is the set of initial ranges of rates for each continuous variable.

The *preset* of a transition t (denoted $\bullet t$) represents the set of places feeding t (i.e., $\bullet t = \{p \mid (p, t) \in F\}$). The *postset* of a transition t (denoted $t\bullet$) represents the set of places that t feeds (i.e., $t\bullet = \{p \mid (t, p) \in F\}$).

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. HSL is an extension of *separation logic* [26] (sometimes referred to as *difference logic*) that allows for non-unit slopes on the separation predicates. These formulas satisfy the following grammar:

$$\phi ::= \textbf{true} \mid \textbf{false} \mid b_i \mid \neg\phi \mid \phi \wedge \phi \mid c_1 x_1 \geq c_2 x_2 + c_3$$

where b_i are Boolean variables, x_1 and x_2 are continuous variables, and c_1, c_2, and c_3 are rational constants in $\mathbb{Q}$. Note that any inequality between two real variables can be formed with $\geq$ and negations of $\geq$ inequalities. Each transition $t \in T$ is labeled using the functions defined in $L = \langle En, D, BA, VA, RA \rangle$:

- $En : T \to \phi$ labels each transition $t \in T$ with an enabling condition;
- $D : T \to |\mathbb{Q}| \times (|\mathbb{Q}| \cup \{\infty\})$ labels each transition $t \in T$ with a lower and upper bound $[d_{\mathrm{l}}, d_{\mathrm{u}}]$ on the delay for t to fire;
- $BA : T \to 2^{(B \times \{0,1\})}$ labels each transition $t \in T$ with Boolean assignments made when t fires;
- $VA : T \to 2^{(V \times \mathbb{Q} \times \mathbb{Q})}$ labels each transition $t \in T$ with a continuous variable assignment range, consisting of a lower and upper bound $[a_l, a_u]$, that is made when t fires;
- $RA : T \to 2^{(V \times \mathbb{Q} \times \mathbb{Q})}$ labels each transition $t \in T$ with a range of rates, consisting of a lower and upper bound $[r_l, r_u]$, that are assigned when t fires.

The LHPN shown in Figure 3 is automatically generated from the VHDL-AMS model in Figure 2. This model tracks the real quantity *Vout* that represents the output voltage. The **if-use** statement is compiled into the LHPN in Figure 3a. The **process** statement is compiled into the LHPN in Figure 3b. Initially *Vout* is -1000 mV and increasing between 18 and 22 mV/μs. After 100 μs, *Vin* is assigned to **true** by the **assign** function which causes *Vout* to begin decreasing at a rate of -22 to -18 mV/μs. The **assert** statement is used to check if *Vout* falls below -2000 mV or goes above 2000 mV and is compiled into the LHPN shown in Figure 3c which fires a transition to set the Boolean signal *fail* to true when the assertion is violated.

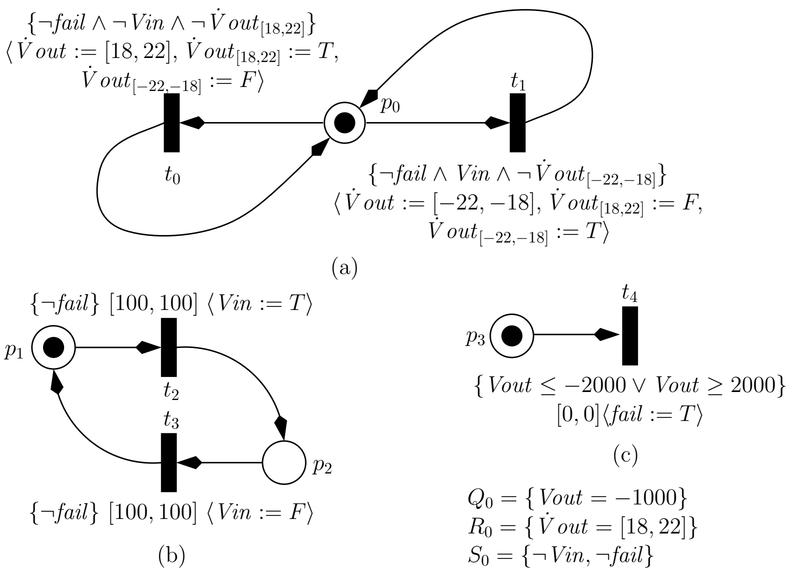

$\{\neg fail \wedge \neg Vin \wedge \neg \dot{V}out_{[18,22]}\}$
$\langle \dot{V}out := [18,22], \dot{V}out_{[18,22]} := T,$
$\dot{V}out_{[-22,-18]} := F\rangle$
t_1
p_0
t_0
$\{\neg fail \wedge Vin \wedge \neg \dot{V}out_{[-22,-18]}\}$
$\langle \dot{V}out := [-22,-18], \dot{V}out_{[18,22]} := F,$
$\dot{V}out_{[-22,-18]} := T\rangle$
(a)
$\{\neg fail\}$ $[100,100]$ $\langle Vin := T\rangle$
p_1
t_2
t_3
p_2
$\{\neg fail\}$ $[100,100]$ $\langle Vin := F\rangle$
(b)
t_4
p_3
$\{Vout \leq -2000 \vee Vout \geq 2000\}$
$[0,0]\langle fail := T\rangle$
(c)
$Q_0 = \{Vout = -1000\}$
$R_0 = \{\dot{V}out = [18,22]\}$
$S_0 = \{\neg Vin, \neg fail\}$

Fig. 3. LHPN of the switched capacitor integrator generated from VHDL-AMS

4 Symbolic Model of LHPNs

In order for analysis to proceed, a symbolic model is generated from the LHPN 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*. Before constructing the symbolic model, a set of real variables and two additional sets of Boolean variables are created in addition to the sets defined for an LHPN. The set of real variables, C, are used to track the values of the clocks on each transition. The *transition clock* for transition t is denoted by c_t. The first set of Boolean variables are known as *clock active variables*, A, and are used to keep track of whether or not the clocks on transitions are active. The clock active variable for transition t is denoted by a_t. The second set of Boolean variables are known as *Boolean rate variables*, *BR*, used for determining the current rate of change for each continuous variable. Boolean rate variables are denoted by $\dot{v}_{[r_l,r_u]}$ for the variable corresponding to the continuous variable v currently advancing at a range of rates $[r_l, r_u]$.

The invariant ($\phi_{\mathcal{I}}$) is an HSL statement that must be satisfied in every state of the system and is calculated as shown in Equation 1.

$$\phi_{\mathcal{I}} = \Phi \wedge \bigwedge_{t \in T} (a_t \Rightarrow \bullet t \wedge En(t) \wedge 0 \le c_t \le d_u(t)) \wedge (\overline{a_t} \Rightarrow \overline{\bullet t} \vee \widetilde{En(t)}) \qquad (1)$$

The invariant first states that only the reachable discrete states (represented by Φ) are allowed. The formula Φ is found by performing a state space exploration of the LHPN while neglecting the continuous variables. The discrete state space exploration is based on the Petri net algorithm described in [27] with extensions to include values of Boolean signals and Boolean rate variables in the state space [25]. In other words, Φ is a formula over the Boolean variables for the Petri net marking, Boolean signals, and Boolean rate variables.

After calculating the discrete state space, Φ, the next step in constructing the system invariant, $\phi_{\mathcal{I}}$, is to insert known information about the continuous state space. This is performed using the clock active variables. Specifically, 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 (denoted $\overline{\bullet t}$) or the *non-strict inverse* ($\widetilde{En(t)}$) of the enabling condition must be satisfied. In the non-strict inverse, all $\geq$ separation predicates become $\leq$ separation predicates and vice-versa. For example, the non-strict inverse of the HSL formula $a \wedge x \leq 2000$ is $\overline{a} \vee x \geq 2000$. The non-strict inverse is used to allow for the existence of a time of overlap when a clock is both allowed to be active and inactive at which time the clock's state can change. 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 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 Φ, the Boolean state set, by existentially abstracting all non-rate Boolean variables. Each product term in Φ corresponds to a ϕ_R of a pair in $\mathcal{R}$.

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*, ϕ_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 apply the assignments associated with t. Formally, the set of primary guarded commands is defined as follows:

$$\mathcal{C}_P = \{\langle \phi_{G_P}(t), \mathcal{A}_P(t)\rangle \mid t \in T\} \tag{2}$$

where $\phi_{G_P}(t) = (\bullet t \wedge \overline{t\bullet - \bullet t} \wedge En(t) \wedge a_t \wedge c_t \geq d_l(t))$ 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 Figure 3 is:

$$\begin{aligned}
\phi_{G_P}(t_2) &= p_1 \wedge \overline{p_2} \wedge \overline{fail} \wedge a_{t_2} \wedge c_{t_2} \geq 100 \\
\mathcal{A}_P(t_2) &= \{p_1 := F, p_2 := T, Vin := T, \\
&\qquad a_{t_2} := F, c_{t_2} := [-\infty, \infty]\}
\end{aligned}$$

Two secondary guarded commands are created for each transition $t \in T$. 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 removes the clock from the state space. The set of secondary guarded commands is defined as follows:

$$\mathcal{C}_S = \{\langle \phi_{G_{SA}}(t), \mathcal{A}_{SA}(t)\rangle, \langle \phi_{G_{SD}}(t), \mathcal{A}_{SD}(t)\rangle \mid t \in T\} \tag{3}$$

where $\phi_{G_{SA}}(t) = \bullet t \wedge En(t) \wedge \overline{a_t}$, $\mathcal{A}_{SA}(t) = \{a_t := T, c_t := [0, 0]\}$, $\phi_{G_{SD}}(t) = (\overline{\bullet t} \vee \overline{En(t)}) \wedge a_t$, and $\mathcal{A}_{SD}(t) = \{a_t := F, c_t := [-\infty, \infty]\}$. The activating and deactivating guarded commands for transition t_1 in Figure 3 are:

$$\begin{aligned}
\phi_{G_{SA}}(t_1) &= p_0 \wedge \overline{fail} \wedge Vin \wedge \overline{\dot{V}out_{[-22,-18]}} \wedge \overline{a_{t_1}} \\
\mathcal{A}_{SA}(t_1) &= \{a_{t_1} := T, c_{t_1} := [0, 0]\} \\
\phi_{G_{SD}}(t_1) &= (\overline{p_0} \vee fail \vee \overline{Vin} \vee \dot{V}out_{[-22,-18]}) \wedge a_{t_1} \\
\mathcal{A}_{SD}(t_1) &= \{a_{t_1} := F, c_{t_1} := [-\infty, \infty]\}
\end{aligned}$$

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. Due to space limitations, only a brief description of the merging process is given. A complete algorithm is described in [25]. 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 [25]. 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. Note that after performing the merge operation, secondary guarded commands whose guards contain inequalities are inserted into the final guarded command set. This is necessary because as time moves forward, the secondary guarded commands could become enabled and cause clocks to be activated or deactivated. However, before the secondary guarded commands are added, their guards must be modified to enforce the threshold on the continuous variables. For example, consider a situation where a transition has the enabling condition $x \geq 5$. The clock on this transition can be activated either when its preset becomes marked when x is already greater than or equal to five, or by x becoming equal to five while the preset is already marked. The first case is handled by the merged guarded command while the second case should be handled by a secondary guarded command that ensures that x is equal to five and continues to increase above five, i.e., when $x \geq 5 \wedge x \leq 5 \wedge \mathtt{incr}(x)$ where $\mathtt{incr}(x)$ returns the disjunction of the Boolean rate variables where the rates are increasing. Similarly, $\mathtt{decr}(x)$ returns the disjunction of the Boolean rate variables where the rates for x are decreasing. In the integrator example, since t_2 assigns Vin to true and marks p_2, it activates the clocks for t_1 and t_3. This results in the following merged guarded command:

$$\begin{aligned}
\phi_G &= p_0 \wedge p_1 \wedge \overline{p_2} \wedge \overline{fail} \wedge \overline{\dot{V}out_{[-22,-18]}} \wedge \overline{a_{t_1}} \wedge a_{t_2} \wedge \overline{a_{t_3}} \wedge c_{t_2} \geq 100 \\
\mathcal{A} &= \{p_1 := F, p_2 := T, Vin := T, \\
&\quad a_{t_1} := T, c_{t_1} := [0,0], a_{t_3} := T, c_{t_3} := [0,0], \\
&\quad a_{t_2} := F, c_{t_2} := [-\infty, \infty]\}
\end{aligned}$$

5 SMT Based Bounded Model Checking

The basic algorithm for performing SMT based bounded model checking of LHPNs is shown in Figure 4. The algorithm proceeds by creating an SMT instance in which statements are asserted. The first step is to create a set of state variables for each iteration of the exploration. The state variables for each iteration, i, are defined using the tuple $\langle M^i, S^i, Q^i, C^i, A^i, BR^i \rangle$. The next step is to assert the initial state (ϕ_{init}) in terms of the initial iteration's variables (i.e., $i = 0$). At this point, the SMT formula is constructed one iteration at a time. For each iteration, it is necessary to assert the invariant in terms of that iteration's set of state variables. Then, each iteration's next states are calculated by firing transitions or elapsing time. This is performed by asserting a disjunction of the guarded commands and a time elapse formula. Finally, a failure condition is asserted in terms of state variables from each iteration. After asserting each of these components, the SMT satisfiability check is performed. Satisfiability indicates that the property is violated because there is an assignment indicating that the failure condition is reachable. Unsatisfiability indicates that the property could not be violated in that number of iterations. This does not necessarily indicate that the property cannot be violated, however, so this is a bounded model checker.

```
smtCheck($\phi_{init}, \phi_{\mathcal{I}}, \mathcal{C}, \mathcal{R}, maxIterations$)
    SMTInstance ins($maxIterations$);
    $i = 0$
    ins.assert($\phi^0_{init}$)
    while ($i$ < $maxIterations$)
        ins.assert($\phi^i_{\mathcal{I}}$)
        $trans = \textbf{true}$
        for each $\langle \phi_G, \mathcal{A} \rangle \in \mathcal{C}$
            $trans = trans \vee$ mkExprForGC($\phi_G, \mathcal{A}, i, i+1$)
        $trans = trans \vee$ mkExprForTimeElapse($\mathcal{R}, i, i+1$)
        ins.assert($trans$)
        $i++$
    ins.assert(mkExprForFailProp($maxIterations$))
    if (ins.check == true) then return ``Property Violated''
    else return ``Property Not Violated''
```

Fig. 4. Algorithm for SMT based bounded model checking

The remainder of this section describes the SMT based bounded model checking algorithm in greater detail.

An assertion for the next state calculation is made based on the disjunction of each guarded command and the time elapse assertion. The transition relation portion of the next state assertion makes use of the merged guarded command set ($\mathcal{C}$) to calculate the values of the next state variables based on the values of the current iteration, i, variables. The algorithm for constructing the assertion statement for a given guarded command is shown in Figure 5. Essentially, the guard portion (ϕ_G) of the guarded command is asserted in terms of the current state while the assignment portion of the guarded command makes use of both the current and the next iteration variables. Assignments that are specified in the assignment set ($\mathcal{A}$) are performed on the next iteration variables while variables that have no assignment performed on them are simply assigned the same value as in the current iteration. There is one exception, however. If a clock is assigned the range $[-\infty, \infty]$, no assignment is made to that clock variable. This allows the clock to remain undefined in the next iteration. In the integrator example, the SMT assertion statement for the merged guarded command that fires t_2 and activates the clocks for t_1 and t_3, given the current iteration i and the next iteration j, is:

$$
\begin{aligned}
&p_0^i \wedge p_1^i \wedge \overline{p_2^i} \wedge \overline{fail^i} \wedge \overline{\dot{V}out_{[-22,-18]}} \wedge \overline{a^i_{t_1}} \wedge a^i_{t_2} \wedge \overline{a^i_{t_3}} \wedge c^i_{t_2} \geq 100 \wedge \\
&p_0^j = p_0^i \wedge p_1^j = \textbf{false} \wedge p_2^j = \textbf{true} \wedge p_3^j = p_3^i \wedge Vin^j = \textbf{true} \wedge fail^j = fail^i \wedge \\
&a^j_{t_0} = a^i_{t_0} \wedge a^j_{t_1} = \textbf{true} \wedge a^j_{t_2} = \textbf{false} \wedge a^j_{t_3} = \textbf{true} \wedge a^j_{t_4} = a^i_{t_4} \wedge \\
&\dot{V}out^j_{[18,22]} = \dot{V}out^i_{[18,22]} \wedge \dot{V}out^j_{[-22,-18]} = \dot{V}out^i_{[-22,-18]} \wedge \\
&c^j_{t_0} = c^i_{t_0} \wedge c^j_{t_1} = 0 \wedge c^j_{t_3} = 0 \wedge c^j_{t_4} = c^i_{t_4} \wedge Vout^j = Vout^i \wedge \delta^{i,j} = 0
\end{aligned}
$$

```
mkExprForGC($\phi_G$, $\mathcal{A}$, $i$, $j$)
    $result = \phi_G^i$          // Guard in terms of current iteration variables.
    foreach $b \in \{M \cup S \cup A \cup BR\}$      // Perform Boolean assignments.
        if (($b :=$ true) $\in \mathcal{A}$) then $result = result \wedge (b^j =$ true)
        else if (($b :=$ false) $\in \mathcal{A}$) then $result = result \wedge (b^j =$ false)
        else $result = result \wedge (b^j = b^i)$
    foreach $v \in \{C \cup Q\}$                  // Perform real assignments.
        if (($v := [-\infty, \infty]$) $\in \mathcal{A}$) then  // Do Nothing.
        else if (($v := [a_l, a_u]$) $\in \mathcal{A}$) then
            $result = result \wedge (v^j \geq a_l) \wedge (v^j \leq a_u)$
        else
            $result = result \wedge (v^j = v^i)$
    $result = result \wedge (\delta^{i,j} = 0)$          // No time advancement
    return $result$
```

Fig. 5. Algorithm to generate an SMT statement for a guarded command

Note that $c_{t_2}^j$ is not assigned any value, since it is to be assigned the value $[-\infty, \infty]$. By not performing any assignment on $c_{t_2}^j$, it can take any value.

The time elapse portion of the next state assertion makes use of the possible rate set ($\mathcal{R}$) to calculate the values of real variables as a result of time moving forward. This algorithm is shown in Figure 6. In calculating the next state via time elapse, a new real variable is created representing the amount of time that has elapsed. This variable is referred to as $\delta^{i,j}$, and it represents the amount of time that has elapsed between iterations i and j. Since time is moving forward, $\delta^{i,j}$ is always greater than or equal to zero. All clock variables increase by exactly $\delta^{i,j}$. Next, based on the current values of the Boolean rate variables, the real variables change by some multiple of $\delta^{i,j}$. Lastly, all Boolean variables in the next iteration have the same value as in the current iteration. The complete time elapse assertion for the integrator, given the current iteration i and next iteration j, is:

$$
\begin{aligned}
&\delta^{i,j} \geq 0 \wedge c_{t_0}^j = c_{t_0}^i + \delta^{i,j} \wedge c_{t_1}^j = c_{t_1}^i + \delta^{i,j} \wedge c_{t_2}^j = c_{t_2}^i + \delta^{i,j} \wedge \\
&c_{t_3}^j = c_{t_3}^i + \delta^{i,j} \wedge c_{t_4}^j = c_{t_4}^i + \delta^{i,j} \wedge \\
&((\dot{V}out_{[18,22]}^i \wedge \overline{\dot{V}out_{[-22,-18]}^i} \wedge 18\delta^{i,j} + Vout^i \leq Vout^j \leq 22\delta^{i,j} + Vout^i) \vee \\
&(\overline{\dot{V}out_{[18,22]}^i} \wedge \dot{V}out_{[-22,-18]}^i \wedge -22\delta^{i,j} + Vout^i \leq Vout^j \leq -18\delta^{i,j} + Vout^i)) \wedge \\
&p_0^j = p_0^i \wedge p_1^j = p_1^i \wedge p_2^j = p_2^i \wedge p_3^j = p_3^i \wedge Vin^j = Vin^i \wedge fail^j = fail^i \wedge \\
&a_{t_0}^j = a_{t_0}^i \wedge a_{t_1}^j = a_{t_1}^i \wedge a_{t_2}^j = a_{t_2}^i \wedge a_{t_3}^j = a_{t_3}^i \wedge a_{t_4}^j = a_{t_4}^i \wedge \\
&\dot{V}out_{[18,22]}^j = \dot{V}out_{[18,22]}^i \wedge \dot{V}out_{[-22,-18]}^j = \dot{V}out_{[-22,-18]}^i
\end{aligned}
$$

The last step in the construction of the SMT formula is to assert that the property is violated (i.e., *fail* becomes true during some iteration). This is

```
mkExprForTimeElapse($\mathcal{R}, i, j$)
    $result = \delta^{i,j} \geq 0$
    foreach $c \in C$  // Increment all clocks by $\delta$
        $result = result \wedge (c^j = c^i + \delta^{i,j})$
    $rates =$ false
    foreach $\langle \phi_R, R \rangle \in \mathcal{R}$  // Increment real variables based on $\mathcal{R}$
        $rate = \phi_R$
        foreach $(\dot{v} := [r_l, r_u]) \in R$
            $rate = rate \wedge (v^j \geq v^i + r_l\delta^{i,j}) \wedge (v^j \leq v^i + r_u\delta^{i,j})$
        $rates = rates \vee rate$
    foreach $b \in \{M \cup S \cup A \cup BR\}$  // Boolean variables stay same value
        $result = result \wedge (b^j = b^i)$
    $result = result \wedge rates$
    return $result$
```

Fig. 6. Algorithm to generate an SMT statement for the time elapse calculation

accomplished by constructing a disjunction of the *fail* variables over all iterations. For five iterations of the integrator example, the result is:

$$fail^0 \vee fail^1 \vee fail^2 \vee fail^3 \vee fail^4$$

The final step of the model checker is to apply the SMT checking procedure. If a satisfiable solution is found, this indicates that it is possible to reach the violating condition. In this event, the SMT solver generates a satisfying solution to the current context. This solution corresponds to a trace over all iteration's state variables beginning from the initial state to the error condition. Since this is a bounded model checker, if the property is not violated within the specified number of iterations, the property may still be violated after more iterations.

6 Results

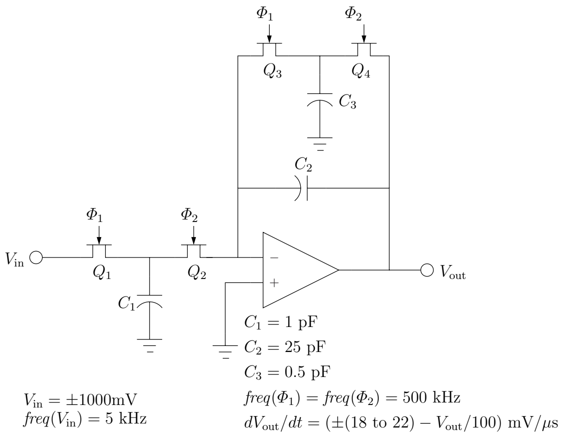
The VHDL-AMS to LHPN compiler, the symbolic model generator, and the SMT bounded model checker have been implemented within the LEMA tool. This section compares the SMT model checker with BDD and DBM model checkers within LEMA. All results use a 2Ghz Intel CoreDuo with 2GB of memory.

The results for the integrator are shown in the top part of Table 1 in which the ranges of rate for the change of *Vout* are varied. In particular, when the lower and upper bound for these rates are equal, all three model checkers determine in a few seconds that the property is satisfied (i.e., the circuit does not saturate). Results for the SMT model checker are presented for both 10 and 20 iterations. When the lower and upper bounds are not equal, both SMT and BDD model checkers find a violation of the property. For example, if the rising slew rate of *Vout* is consistently larger than the falling slew rate, there can be a build up of charge leading to saturation of *Vout*. Note that the DBM model checker cannot directly support ranges of rates. Therefore, a piecewise approximate model must

Table 1. Switched capacitor integrator verification results

Example	Exp. Result	SMT Time (s)	SMT Iter.	BDD Time (s)	BDD Iter.	DBM Time (s)	DBM Zones
Original ([20, 20])	Pass	< 1	10	< 1	7	< 1	4
Original ([20, 20])	Pass	7	20	–	–	–	–
Original ([18, 22])	Fail	< 2	15	< 2	11	n/a	n/a
Piecewise ([18, 22])	Fail	60	20	< 1	6	< 1	9
Corrected	Pass	28	10	6*	6*	n/a	n/a
Corrected	Pass	388	20	–	–	–	–
Corrected piecewise	Pass	249	10	OOM	3	< 1	54
Corrected piecewise	Pass	980	20	–	–	–	–

* Verification result does not match expected result.



Φ_1
Φ_2
Q_3
Q_4
C_3
C_2
Φ_1
Φ_2
V_{in}
Q_1
Q_2
C_1
V_{out}
$C_1 = 1$ pF
$C_2 = 25$ pF
$C_3 = 0.5$ pF
$V_{\text{in}} = \pm 1000$mV
$freq(V_{\text{in}}) = 5$ kHz
$freq(\Phi_1) = freq(\Phi_2) = 500$ kHz
$dV_{\text{out}}/dt = (\pm(18 \text{ to } 22) - V_{\text{out}}/100)$ mV/μs

Fig. 7. Circuit diagram of a corrected switched capacitor integrator

first be generated in which the rate of *Vout* initially increases at 18 mV/μs. After some random amount of time, the rate may switch to 22 mV/μs. Decreasing rates for *Vout* are modeled in a similar way.

Saturation of the integrator can be prevented using the circuit shown in Figure 7. This circuit uses a switched capacitor resistor inserted in parallel with the feedback capacitor to cause *Vout* to drift back to 0 V. In other words, if *Vout* is increasing, it increases faster below 0 V than above. In this circuit's model, the range for *Vout* is 28 to 37 mV/μs when below -1000 mV, 18 to 32 mV/μs when below 0 mV, 8 to 22 mV/μs when below 1000 mV, and 3 to 12 mV/μs when above 1000 mV. Similar rates are used when *Vout* is decreasing. The verification results for the corrected integrator are shown in the bottom part of Table 1. The SMT model checker correctly determines that this circuit does not violate the property for 10 and 20 iterations. For this model, the BDD model checker finds a failure erroneously. This false negative is due to inexactness that results from not adding transitivity constraints at all necessary phases of the the analysis. If transitivity constraints are added at each step, BDD analysis quickly runs out of memory. Since the DBM model checker does not support ranges of rates directly, it cannot be applied to this model. Again, an approximate piecewise model can be verified by the DBM model checker. Ironically, the SMT model checker performs better on the more accurate model, since the added transitions in the piecewise model significantly increase the complexity of the SMT formula.

7 Conclusions

This paper describes an SMT bounded model checking algorithm for AMS circuits. These circuits can be described using VHDL-AMS and automatically compiled into an LHPN representation for analysis. This LHPN model is translated into a symbolic model composed of an invariant, possible rates set, and guarded commands. This symbolic model is then automatically converted into an SMT formula for a given number of iterations. If this SMT formula is satisfiable, the satisfying assignment represents an error trace for the circuit being verified.

One promising abstraction and refinement approach is to combine the BDD and SMT model checkers. The BDD model checker is capable of performing an unbounded full state space exploration, but it often runs out of memory due to the large number of BDD variables created. The SMT model checker efficiently determines if the full model violates the property, but it can never guarantee that the property is not violated. Therefore, the BDD model checker could be applied to an abstract model. If the BDD model checker determines that the property is violated in the abstract model, the SMT model checker can be used with the full model to ensure that the failure is not a false negative. In this case, the BDD model checker would specify the number of iterations that are required for the abstract model to fail. If the SMT model checker verifies that the full model does fail, verification is complete. If the full model does not violate the property, the violation is a false negative and the unsatisfying core can be used to refine the abstract model. This process repeats until a true failure is found or the BDD model checker determines that the abstract model does not violate the property. Another interesting approach to consider would be to apply the proof-based iterative abstraction and refinement method from [28].

References

1. Kurshan, R.P., McMillan, K.L.: Analysis of digital circuits through symbolic reduction. IEEE Transactions on CAD 10(11), 1356–1371 (1991)
2. Hartong, W., Hedrich, L., Barke, E.: Model checking algorithms for analog verification. In: Proc. of DAC, pp. 542–547 (2002)
3. Gupta, S., Krogh, B.H., Rutenbar, R.A.: Towards formal verification of analog designs. In: Proc. of ICCAD, pp. 210–217 (2004)
4. Dang, T., Donze, A., Maler, O.: Verification of analog and mixed-signal circuits using hybrid systems techniques. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 21–36. Springer, Heidelberg (2004)

5. Frehse, G., Krogh, B.H., Rutenbar, R.A.: Verifying analog oscillator circuits using forward/backward refinement. In: Proc. of DATE, pp. 257–262 (2006)
6. Little, S., Seegmiller, N., Walter, D., Myers, C.J.: Verification of analog/mixed-signal circuits using labeled hybrid petri nets. In: Proc. of ICCAD, pp. 275–282 (2006)
7. Walter, D., Little, S., Seegmiller, N., Myers, C., Yoneda, T.: Symbolic model checking of analog/mixed-signal circuits. In: Proc. of ASPDAC, pp. 316–323 (2007)
8. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
9. Armando, A., Castellini, C., Giunchiglia, E.: SAT-based procedures for temporal reasoning. In: Biundo, S., Fox, M. (eds.) ECP 1999. LNCS, vol. 1809, pp. 97–108. Springer, Heidelberg (2000)
10. Armando, A., Castellini, C., Giunchiglia, E., Maratea, M.: A sat-based decision procedure for the boolean combination of difference constraints. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, Springer, Heidelberg (2005)
11. Barrett, C., Dill, D., Stump, A.: Checking satisfiability of first-order formulas by incremental translation to sat. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, Springer, Heidelberg (2002)
12. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., van Rossum, P., Schulz, S., Sebastiani, R.: An incremental and layered procedure for the satisfiability of linear arithmetic logic. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 317–333. Springer, Heidelberg (2005)
13. de Moura, L., Rue, H.: Lemmas on demand for satisfiability solvers. In: Proc. of SAT (2002)
14. Filliâtre, J.C., Owre, S., Rue, H.: ICS: Integrated Canonization and Solving (Tool presentation). In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 246–249. Springer, Heidelberg (2001)
15. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
16. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using smt solvers instead of sat solvers. In: Valmari, A. (ed.) SPIN 2006. LNCS, vol. 3925, pp. 146–162. Springer, Heidelberg (2006)
17. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Ranise, S., van Rossum, P., Sebastiani, R.: Efficient satisfiability modulo theories via delayed theory combination. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 335–349. Springer, Heidelberg (2005)
18. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and Abstract DPLL Modulo Theories. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 36–50. Springer, Heidelberg (2005)
19. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
20. Myers, C.J., Harrison, R.R., Walter, D., Seegmiller, N., Little, S.: The case for analog circuit verification. Electronic Notes Theoretical Computer Science 153(3), 53–63 (2006)
21. Zheng, H.: Specification and compilation of timed systems. Master's thesis, University of Utah (1998)
22. Myers, C.: Asynchronous Circuit Design. Wiley, Chichester (2001)

23. David, R., Alla, H.: On hybrid petri nets. Discrete Event Dynamic Systems: Theory and Applications 11, 9–40 (2001)
24. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Hybrid Systems, pp. 209–229 (1992)
25. Walter, D.: Verification of Analog and Mixed-Signal Circuits Using Symbolic Methods. PhD thesis, University of Utah (2007)
26. Henzinger, T., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. In: 7th Symposium of Logics in Computer Science, pp. 394–406. IEEE Computer Scienty Press, Los Alamitos (1992)
27. Pastor, E., Roig, O., Cortadella, J., Badia, R.M.: Petri net analysis using boolean manipulation. In: Valette, R. (ed.) PNPM 1994. LNCS, vol. 815, pp. 416–435. Springer, Heidelberg (1994), `citeseer.ist.psu.edu/pastor94petri.html`
28. McMillan, K., Amla, N.: Automatic abstraction without counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)