Verification of Analog/Mixed-Signal Circuits Using Labeled Hybrid Petri Nets[*]

Scott Little
University of Utah
Salt Lake City, UT 84112, USA
little@cs.utah.edu

Nicholas Seegmiller
University of Utah
Salt Lake City, UT 84112, USA
seegmill@cs.utah.edu

David Walter
University of Utah
Salt Lake City, UT 84112, USA
dwalter@cs.utah.edu

Chris Myers
University of Utah
Salt Lake City, UT 84112, USA
myers@vlsigroup.utah.edu

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

ABSTRACT

System on a chip design results in the integration of digital, analog, and mixed-signal circuits on the same substrate which further complicates the already difficult validation problem. This paper presents a new model, *labeled hybrid Petri nets* (LHPNs), that is developed to be capable of modeling such a heterogeneous set of components. This paper also describes a compiler from VHDL-AMS to LHPNs. To support formal verification, this paper presents an efficient zone-based state space exploration algorithm for LHPNs. This algorithm uses a process known as *warping* to allow zones to describe continuous variables that may be changing at variable rates. Finally, this paper describes the application of this algorithm to a couple of analog/mixed-signal circuit examples.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aids—*Verification*

General Terms

Verification, design

Keywords

Formal methods, hybrid Petri nets

1. INTRODUCTION

System on a chip (SoC) designs are becoming common in the marketplace. These designs often include both digital and *analog/mixed-signal* (AMS) circuits that must interact presenting a challenge to traditional validation workflows. Digital validation methods are based on relatively long time steps and simulation times while analog validation uses comparatively short time steps and simulation times. As a result, digital validation methods are typically applied to digital blocks while analog validation methods are applied to analog blocks. Such strict divisions, however, are rarely possible, since blocks validated using analog methods often contain digital components which are inadequately validated during analog verification and vice versa. Therefore, it is necessary to develop new functional validation methods that can support the heterogeneous nature of SoC designs.

Formal verification is a validation method that has shown advantages over simulation based methodologies for digital systems [4]. Based on this success there has recently been some research on applying formal verification based methods to AMS circuits [5, 8–16, 18, 21]. The major challenge in verifying AMS circuits is that continuous values such as voltages and currents must be tracked accurately complicating an already difficult state space exploration process. Hartong, Hedrich, and Barke create a Boolean abstraction for the system and then use standard digital verification methods to do the analysis [12–15]. While a promising approach, this technique loses significant accuracy in the abstraction to a Boolean model. Frehse's `PHAVer` model checker analyzes linear hybrid automata models of AMS circuits using convex polyhedra to represent the continuous state space [8, 9]. While these polyhedra can become quite complex, one unique feature of `PHAVer` is that it allows for performance to be tuned at the expense of a conservative state space. Previously, we developed a tool that leverages efficient verification algorithms for time/timed Petri nets to analyze hybrid Petri net models of AMS circuits using a process known as *warping* that normalizes the advancement of all continuous variables to a rate of one [18].

Crucial to the acceptance of a new formal verification methodology by AMS designers is the ability to specify their circuits of interest using a familiar language. The previous approaches require the designer to describe their AMS circuit using either hybrid automata or hybrid Petri nets. This paper presents a new *labeled hybrid Petri net* (LHPN) model that has been developed such that it can be readily gen-

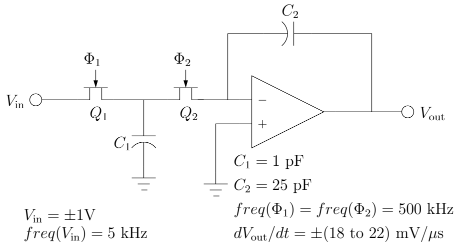
[*] This research is supported by SRC contract 2005-TJ-1357 and an SRC Graduate Fellowship.

ICCAD '06, November 5-9, 2006, San Jose, CA

erated from VHDL-AMS, a standard hardware description language for AMS circuits. To support formal verification, this paper also describes an efficient zone-based state space exploration algorithm for LHPNs. This algorithm uses the warping process described in [18] to allow zones the capability of representing continuous variables that are changing at variable rates. A compiler from VHDL-AMS to LHPNs and a formal verification tool for LHPNs have been implemented. Verification results on AMS circuits are promising.

2. MOTIVATING EXAMPLE

The switched capacitor integrator circuit shown in Fig. 1 is a circuit used as a component in many AMS circuits such as ADCs and DACs. Although only a small piece of these complex circuits, the switched capacitor integrator proves to be a useful example illustrating the type of problems that can be present in AMS circuit designs. Discrete-time integrators typically utilize switched capacitor circuits to accumulate charge. Capacitor mismatch can cause gain errors in integrators. Also, the CMOS switch elements in switched-capacitor circuits inject charge when they transition from closed to open. This charge injection is difficult to control with any precision, and its voltage-dependent nature leads to circuits that have a weak signal-dependent behavior. This can cause integrators to have slightly different gains depending on their current state and input value. Circuits using integrators run the risk of the integrator saturating near one of the power supply rails. It is essential to ensure that this never happens during operation under any possible permutation of component variations. Therefore, the verification property to check for this circuit is whether or not the voltage V_{out} can rise above 2V or fall below -2V. For this example, we assume due to noise and uncertainty in model parameters that the output slew rate has a variance of ±10 percent (i.e., $\pm$(18 to 22) mV/μs). This circuit, therefore, must be verified for all values in the range [20].



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

Figure 1: A switched capacitor integrator.

Fig. 2 depicts a VHDL-AMS description for an integrator. This model tracks the real quantities *Vout*, that represents the output voltage, and *c*, that represents a clock tracking the timing of changes in the input voltage. This model represents the range in output slew rate using a piecewise linear approximation controlled by the Boolean variables *inc*18 and *inc*22. Another Boolean variable, *clkrst*, is used to reset the clock, *c*, at the end of each half cycle. The initial conditions are that *Vout* is -1V (i.e., -1000 mV) and that *c* is 0 indicating the start of the input cycle. The clock *c* increases at a rate of 1 until it reaches the completion of half an input cycle (i.e., 100 μs). At this point, it receives a *clkrst* signal and is reset to 0. The **if-use** statement and **process** statement control the rate of change of *Vout*. The process begins by assigning *inc*18 to '1' (this uses the `assign` function defined in the `handshake` package [19]) which starts *Vout* increasing at a rate of 18 mV/μs. After a randomly selected time period between 0 and 100 μs, it sets *inc*22 to '1' changing the rate of increase to 22 mV/μs. It then waits until the first half cycle has completed indicated by *c* reaching 100 μs. At this point, it resets *c* by toggling the *clkrst* signal. Next, it sets *inc*18 to '0' causing *Vout* to begin decreasing at a rate of 18 mV/μs. The remainder of the process modeling the decrease of *Vout* is similar.

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.nondeterminism.all;
use work.handshake.all;
entity integrator is
end integrator;
architecture switchCap of integrator is
   quantity Vout,c:real;
   signal inc18,inc22,clkrst:std_logic:='0';
begin
   break Vout=> -1000.0, c=> 0.0; --Init vals
   c'dot == 1.0; -- Constant clock rate
   break c=> 0.0 when clkrst and
      c'above(100.0);
   if inc18='0' use
     if inc22='0' use Vout'dot == -22.0;
     else Vout'dot == -18.0;
     end use;
   else
     if inc22='0' use Vout'dot == 18.0;
     else Vout'dot == 22.0;
     end use;
   end use;
   process begin
     assign(inc18,'1',0,0);
     assign(inc22,'1',0,100);
     wait until c'above(100.0);
     assign(clkrst,'1',0,0);
     wait until not c'above(100.0);
     assign(clkrst,'0',0,0);
     assign(inc18,'0',0,0);
     assign(inc22,'0',0,100);
     wait until c'above(100.0);
     assign(clkrst,'1',0,0);
     wait until not c'above(100.0);
     assign(clkrst,'0',0,0);
   end process;
end switchCap;
```

Figure 2: VHDL-AMS for an integrator.

3. LABELED HYBRID PETRI NETS

An LHPN is a Petri net model developed to represent AMS circuits. The model is inspired by features in both hybrid Petri nets [6] and hybrid automata [1]. This model has also been developed in such a way that it can be easily generated from VHDL-AMS descriptions. 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 restricted *hybrid separation logic* (HSL) formulas which are a Boolean combination of Boolean variables and separation predicates (inequalities relating continuous variables and rates to constants). These formulas satisfy the following grammar:

$$\phi \quad ::= \quad \textbf{true} \mid \textbf{false} \mid b_i \mid \neg\phi \mid \phi \wedge \phi \mid v_i \bowtie_i k_i \mid \dot{v}_i \bowtie_i k_i$$

where b_i is a Boolean variable, v_i is a continuous variable, $\dot{v}_i$ is the rate of a continuous variable, k_i is a rational constant from the set of rational numbers $\mathbb{Q}$, and $\bowtie_i$ is a relational operator from the set: $\{\leq, \geq\}$. The labels permitted in LHPNs are represented using a tuple $L = \langle En, D, BA, VA, RA \rangle$:

- $En : T \rightarrow \phi$ labels each transition $t \in T$ with an enabling condition;
- $D : T \rightarrow |\mathbb{Q}| \times (|\mathbb{Q}| \cup \{\infty\})$ labels each transition with a lower and upper bound delay value, $[d_{\mathrm{l}}, d_{\mathrm{u}}]$;
- $BA : T \rightarrow 2^{(B \times \{0,1\})}$ labels each transition $t \in T$ with Boolean assignments made when t fires;
- $VA : T \rightarrow 2^{(V \times \mathbb{Q})}$ labels each transition $t \in T$ with continuous variable assignments made when t fires;
- $RA : T \rightarrow 2^{(V \times \mathbb{Q})}$ labels each transition $t \in T$ with continuous rate assignments made when t fires.

An example LHPN is shown in Fig. 3 which is automatically generated from the VHDL-AMS model in Fig. 2. The VHDL-AMS to LHPN compiler is built using a method similar to the one in [22]. The first two lines of the architecture in Fig. 2 set the initial values for *Vout* and c and sets the rate of change for c to 1. The second **break** statement is compiled into the LHPN shown in Fig. 3(a). This LHPN resets c to 0 when *clkrst* is high and c reaches 100 μs. The **if-use** statement is compiled into the LHPN shown in Fig. 3(b). This LHPN sets the rate of change of *Vout* based upon the current value of the Boolean variables *inc*18 and *inc*22. This net essentially checks the current values of these variables and fires a transition whenever the rate needs to change. The **process** statement is compiled into the LHPN shown in Fig. 3(c). This LHPN performs the assignments to the Boolean signals *inc*18, *inc*22, and *clkrst*. Note that since LHPNs require the rate of each continuous variable to be constant within a discrete state only piecewise linear system models are currently supported.

The state of an LHPN is defined using a 6-tuple of the form $\psi = \langle M, S, Q, R, I, C \rangle$ where:

- $M \subseteq P$ is the set of marked places;
- $S : B \rightarrow \{0, 1\}$ is the value of each Boolean signal;
- $Q : V \rightarrow \mathbb{Q}$ is the value of each continuous variable;
- $R : V \rightarrow \mathbb{Q}$ is the rate of each continuous variable;
- $I : ineq(En) \rightarrow \{0, 1\}$ is the value of each inequality of the form $v_i \bowtie_i k_i$ used in En.
- $C : T \rightarrow \mathbb{Q}$ is the value of each transition clock.

The function $ineq(En)$ returns the set of inequalities of the form $v_i \bowtie_i k_i$ used in transition enabling conditions. The current state of an LHPN can change via a transition firing or time advancement.

Every transition $t \in T$ has a *preset* denoted by $\bullet t = \{p \mid (p,t) \in F\}$ and a *postset* denoted by $t\bullet = \{p \mid (t,p) \in F\}$. A transition $t \in T$ is enabled when all of the places in its preset are marked (i.e., $\bullet t \subseteq M$), and the enabling condition on t evaluates to true (i.e., $Eval(En(t), \psi)$ where the function $Eval$ evaluates an HSL formula on a given state). The function $\mathcal{E}(\psi)$ is defined to return the set of enabled transitions for the given state. When a transition t becomes enabled, its clock is initialized to zero. The transition t can then fire at any time after its clock satisfies its lower delay bound and before it exceeds its upper delay bound (i.e., $d_l(t) \leq C(t) \leq d_u(t)$) as long as it remains continuously enabled. From a state ψ, a new state ψ' can be reached by firing a transition t found in $\mathcal{E}(\psi)$. This new state is defined as follows:

- $M' = (M - \bullet t) \cup t\bullet$;
- $S'(b_i) = \begin{cases} s & \text{if } \exists (b_i, s) \in BA(t) \\ S(b_i) & \text{otherwise} \end{cases}$
- $Q'(v_i) = \begin{cases} x & \text{if } \exists (v_i, x) \in VA(t) \\ Q(v_i) & \text{otherwise} \end{cases}$
- $R'(v_i) = \begin{cases} x & \text{if } \exists (v_i, x) \in RA(t) \\ R(v_i) & \text{otherwise} \end{cases}$
- $I'(i_i) = (Q'(v_i) \bowtie_i k_i)$
- $C'(t_i) = \begin{cases} 0 & \text{if } t_i \in \mathcal{E}(\psi') \wedge t_i \notin \mathcal{E}(\psi) \\ C(t_i) & \text{otherwise} \end{cases}$

In other words, the Boolean, variable, and rate assignments associated with transition t_i are executed, and the clocks associated with newly enabled transitions are reset to 0.

In a state ψ, time can advance by any value τ which is less than $\tau_{\max}(\psi)$. The value of $\tau_{\max}(\psi)$ is the largest amount of time that may pass before a transition is forced to fire (i.e., the clock associated with it exceeds its upper bound) or an inequality changes value (i.e., its continuous variable's value v_i crosses the constant k_i). It is defined as follows:

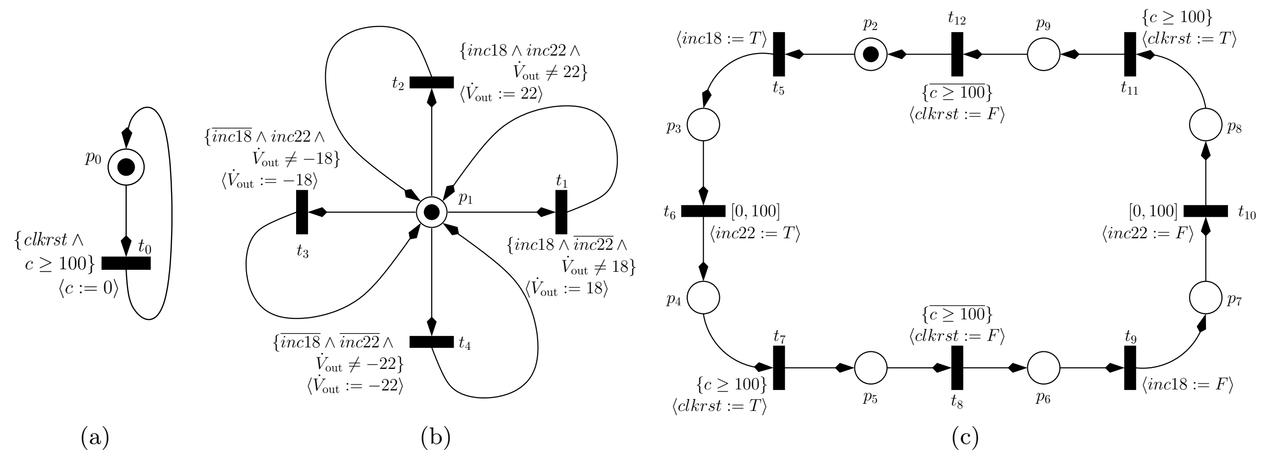
$$\tau_{\max}(\psi) = min \begin{cases} C(t_i) - d_u(t_i) & \forall t_i \in \mathcal{E}(\psi) \\ & \forall v_i \bowtie_i k_i \in ineq(En). \\ \frac{k_i - Q(v_i)}{R(v_i)} & I(i_i) \neq (R(v_i) \bowtie_i 0) \end{cases}$$

The new state after time advancement is defined as follows:

- $Q'(v_i) = Q(v_i) + \tau \cdot R(v_i)$
- $I'(i_i) = \begin{cases} R(v_i) \bowtie_i 0 & \text{if } Q'(v_i) = k_i \\ I(i_i) & \text{otherwise} \end{cases}$
- $C'(t_i) = \begin{cases} 0 & \text{if } t_i \in \mathcal{E}(\psi') \wedge t_i \notin \mathcal{E}(\psi) \\ C(t_i) + \tau & \text{otherwise} \end{cases}$

All other parts of the state are unaffected.

The semantics of the LHPN model are now illustrated using the LHPN example shown in Fig. 3. In the initial state, p_0, p_1, and p_2 are marked; *inc*18, *inc*22, and *clkrst* are false; c is 0 and *Vout* is $-1{,}000$; and the rate of c is 1 and the rate of *Vout* is -22 mV/μs. The only transition enabled in the initial state is t_5 which fires without any time advancement since it has a delay bound of $[0, 0]$. The firing


p_0
t_0
$\{clkrst\ \wedge$
$c \geq 100\}$
$\langle c := 0\rangle$
t_2
$\{inc18 \wedge inc22\ \wedge$
$\dot{V}_{\text{out}} \neq 22\}$
$\langle \dot{V}_{\text{out}} := 22\rangle$
$\{\overline{inc18} \wedge inc22\ \wedge$
$\dot{V}_{\text{out}} \neq -18\}$
$\langle \dot{V}_{\text{out}} := -18\rangle$
p_1
t_1
t_3
$\{inc18 \wedge \overline{inc22}\ \wedge$
$\dot{V}_{\text{out}} \neq 18\}$
$\langle \dot{V}_{\text{out}} := 18\rangle$
t_4
$\{\overline{inc18} \wedge \overline{inc22}\ \wedge$
$\dot{V}_{\text{out}} \neq -22\}$
$\langle \dot{V}_{\text{out}} := -22\rangle$
$\langle inc18 := T\rangle$
p_2
t_{12}
p_9
$\{c \geq 100\}$
$\langle clkrst := T\rangle$
t_5
$\{\overline{c \geq 100}\}$
$\langle clkrst := F\rangle$
t_{11}
p_3
p_8
t_6
$[0, 100]$
$\langle inc22 := T\rangle$
$[0, 100]$
t_{10}
$\langle inc22 := F\rangle$
p_4
p_7
$\{\overline{c \geq 100}\}$
$\langle clkrst := F\rangle$
t_7
t_9
$\{c \geq 100\}$
$\langle clkrst := T\rangle$
p_5
t_8
p_6
$\langle inc18 := F\rangle$

(a) (b) (c)

Figure 3: An LHPN for the switched capacitor integrator example. Note that transitions without a delay bound are assumed to have a [0,0] bound. Also note that $\dot{v}_i \neq k_i$ is a shorthand for $\overline{\dot{v}_i \leq k_i \wedge \dot{v}_i \geq k_i}$.

of t_5 sets *inc*18 to true. When *inc*18 is set to true, transition t_1 becomes enabled and fires immediately changing the rate of *Vout* to 18 mV/μs. At this point, the only enabled transition is t_6 which has a delay bound of $[0, 100]$. Therefore, time can advance from 0 to 100 μs before transition t_6 must fire. When t_6 fires, *inc*22 is set to true. This enables t_2 which fires immediately changing the rate of *Vout* to 22 mV/μs. At the end of the first half of the input cycle (indicated by c reaching 100 μs), the transition t_7 becomes enabled and fires immediately setting *clkrst* to true. This enables transition t_0 which fires immediately resetting c to 0. This enables transition t_8 which fires immediately setting *clkrst* back to false. Operation continues in this manner for the decreasing period of *Vout*.

4. ANALYSIS OF LHPNS

To analyze and verify properties of LHPNs, state space exploration needs to be performed. This analysis is complicated by the fact that LHPNs typically have an infinite number of states. Therefore, to perform state space exploration on LHPNs, it is necessary to represent this infinite number of states using a finite number of state equivalence classes called *state sets*. Our analysis method uses *zones* defined using *difference bound matrices* (DBMs) [7] to represent the continuous portion of the state space. Our methodology is based upon one for analyzing timed systems [2,3] with extensions necessary to deal with continuous quantities changing at variable rates. The state sets are represented with the tuple $\psi = \langle M, S, Q, R, I, Z\rangle$ where:

- $M \subseteq P$ is the set of marked places;
- $S : B \rightarrow \{0, 1\}$ is the value of each Boolean signal;
- $Q : V \rightarrow \mathbb{Q} \times \mathbb{Q}$ is the value of each inactive variable;
- $R : V \rightarrow \mathbb{Q}$ is the rate of each continuous variable;
- $I : ineq(En) \rightarrow \{0, 1\}$ is the value of each inequality of the form $v_i \bowtie_i k_i$ used in En;
- $Z : (T \cup V \cup \{c_0\}) \times (T \cup V \cup \{c_0\}) \rightarrow \mathbb{Q}$ is a DBM composed of active transition clocks, active continuous variables, and c_0 (a reference clock which is always 0).

State sets and states differ in two ways. First, a DBM Z represents the ranges of values for clocks and active continuous variables. Second, inactive continuous variables (i.e., $R(v_i) = 0$) may also have a range of values. The algorithms below use the notation $x \in Z$ to check if the DBM Z is defined for x where x is a transition or continuous variable. The notation $\texttt{nlb}(Z, x)$ accesses the negative of the lower bound for x in Z and $\texttt{ub}(Z, x)$ accesses its upper bound.

The DBM based method for state space exploration of LHPNs is shown in Algorithm 1. The algorithm is a depth first search of the state space. The algorithm begins by constructing the initial state set for the LHPN and adding it to the set of reachable state sets, Ψ. In the initial state set, $M = M_0$, $S = S_0$, and $R = R_0$. Initially, Q includes inactive continuous variables set to their initial value, Q_0. The DBM Z includes active continuous variables (i.e., $R_0(v_i) \neq 0$) set to their initial value and clocks for enabled transitions set to zero. Finally, I contains the initial value for all inequalities (i.e., $I(i_i) = (Q_0(v_i) \bowtie_i k_i)$). The algorithm then calls `findPossibleEvents` which determines all possible events that can result in a new state set. Given this set of events, E, an event, e, is arbitrarily chosen by the `select` function. After the selection of an event, if events still remain in E, the remaining events and the current state set are pushed onto the stack. Given the current state set, ψ, and possible event, e, the state set is updated. If the resulting state set has not been seen before then it is added to the state space, new events are found, and the loop continues. If the state set has been seen before, the stack is popped and the loop continues. If the stack is empty, the reachable state space has been found and is returned. The remainder of this section explains these steps in more detail.

The `findPossibleEvents` algorithm shown in Algorithm 2 determines which events are possible from the current state. There are two types of possible events: a transition firing or an inequality changing value due to the advancement of time. A transition may fire anytime after the lower bound of the delay for that transition has been reached, and it must fire before the upper bound of the delay is exceeded. Clocks are activated when a transition becomes enabled and only enabled transitions are in Z. Therefore, any transition in Z whose clock can reach its lower bound may fire (i.e.,

Algorithm 1: reach()

```
$\psi$ = initialStateSet();
$\Psi = \{\psi\}$;
$E$ = findPossibleEvents($\psi$);
while $true$ do
    $e$ =select($E$);
    if $E - \{e\} \neq \emptyset$ then push($E - \{e\}, \psi$);
    $\psi'$ = updateState($\psi, e$);
    if $\psi' \notin \Psi$ then
        $\Psi = \Psi \cup \{\psi'\}$;
        $\psi = \psi'$;
        $E$ = findPossibleEvents($\psi$);
    else
        if $stack\ not\ empty$ then $(E, \psi)$ = pop();
        else return $\Psi$;
```

$\texttt{ub}(Z, t) \geq d_l(t)$). An inequality may change value when time can advance to the point where the value of the continuous variable associated with the inequality crosses the constant in the inequality. This is determined by the `ineqCanChange` function by examining the current values of R, I, and Z. Once an event has been selected as a possible event, the `addSetItem` function is called to determine if this event can be the next to occur. There are two possible outcomes. The first outcome is that the newly found event cannot actually happen before some other event already in the set E, and it is not added to the set. The second outcome is that the newly found event can occur next, so it is added in the event set, E. Adding it to this set may cause other events already in E to need to be removed from E as this newly added event may occur before previously added events.

Algorithm 2: findPossibleEvents(ψ)

```
$E = \emptyset$;
forall $t \in Z$ do
    if ub($Z, t$) $\geq d_l(t)$ then
        $E$ = addSetItem($E, t$);
forall $i \in ineq(En)$ do
    if ineqCanChange($R, I, Z, i$) then
        $E$ = addSetItem($E, i$);
return $E$;
```

Algorithm 3 updates the state set, ψ, as a result of an event, e. The algorithm begins by calling the `restrict` function to modify Z to reflect that time must have advanced to the point necessary for the event to have occurred (i.e., the clock for the transition firing reaches its lower bound, or the continuous variable reaches the constant in the inequality changing value). Next, the `recanonicalize` function is called to apply Floyd's all-pairs shortest path algorithm to restore Z to a canonical form. When an inequality changes, the next step is to simply update the value of I. When a transition fires, however, the state update required is a bit more involved as shown in Algorithm 4 described below. Next, the transitions are checked to see if they have become newly enabled or disabled. A clock for a transition t not in Z that is enabled must be added to Z while a clock for a transition t in Z that is not enabled must be removed from Z. Finally, time is advanced using Algorithm 5, Z is recanonicalized again, and the new state set is returned.

Algorithm 3: updateState(ψ, e)

```
$Z$ = restrict($Z, e$);
$Z$ = recanonicalize($Z$);
if $e \notin T$ then
    $\psi$ = updateIneq($\psi, e$);
else
    $\psi$ = fireTransition($\psi, e$);
forall $t \in T$ do
    if $t \notin Z \wedge t \in \mathcal{E}(\psi)$ then
        $Z$ = addT($Z, t$);
    else if $t \in Z \wedge t \notin \mathcal{E}(\psi)$ then
        $Z$ = rmT($Z, t$);
$Z$ = advanceTime($Z, R, I$);
$Z$ = recanonicalize($Z$);
return $\psi$;
```

The `updateState` function calls Algorithm 4 to fire a transition t in state set ψ. This algorithm first updates the marking by removing the tokens from all places in $\bullet t$ and adding tokens to all places in $t\bullet$. Next, the transition t must be removed from Z. Then, all assignments labeled on t must be performed. This includes variable assignments, rate assignments, and Boolean signal assignments. The rate assignments may have activated or deactivated a continuous variable, so all continuous variables are checked and added or removed from Z as necessary. Finally, Z is warped using Algorithm 6 to properly account for any rate changes that may have occurred.

Algorithm 4: fireTransition(ψ, t)

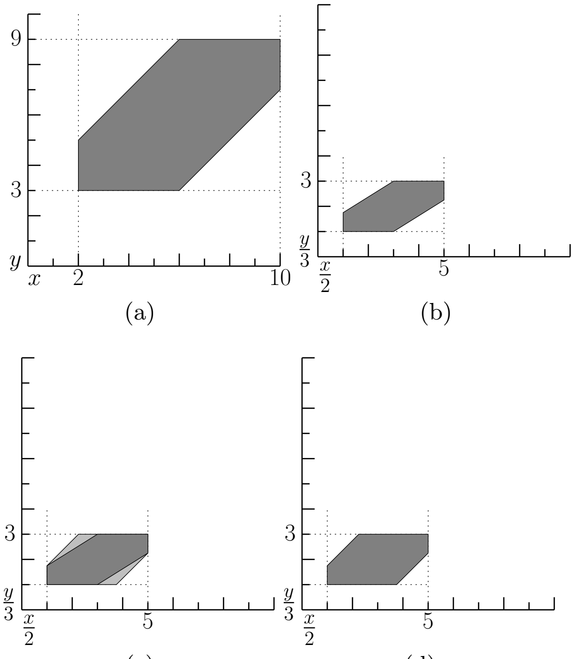
```
$M = (M - \bullet t) \cup t\bullet$;
$Z$ = dbmRemove($Z$,$t$);
$(Z, Q)$ = doVarAsgn($Z, Q, VA(t)$);
$R'$ = doRateAsgn($R, RA(t)$);
$S$ = doBoolAsgn($S, BA(t)$);
forall $v \in V$ do
    if $v \notin Z \wedge R'(v) \neq 0$ then
        $(Z, Q)$ = addV($Z, Q, v$);
    else if $v \in Z \wedge R'(v) = 0$ then
        $(Z, Q)$ = rmV($Z, Q, v$);
$(Z, R)$ = dbmWarp($Z, R, R'$);
return $\psi$;
```

The `updateState` function calls Algorithm 5 to advance time in Z. The basic idea is that time should be allowed to advance as far as possible without missing an event. To ensure that the firing of an enabled transition t is not missed, `advanceTime` sets the upper bound value for the clock associated with t to the upper delay bound for t. To ensure that a change in inequality value is not missed on a variable v, all inequalities involving variable v are checked by the function `checkIneq`, and the largest amount of time that can advance before one of these inequalities changes value is assigned to the upper bound value for v.

State sets for an LHPN cannot be represented exactly in a DBM due to the requirement that all dimensions advance at rate one. While clocks associated with transitions always increase at rate one, continuous variables may increase or decrease with any rate. Therefore, an approximation is necessary to analyze the state space using DBMs. When

```
Algorithm 5: advanceTime($Z, R, I$)
forall $t \in Z$ do
    $\texttt{ub}(Z,t) = d_u(t)$;
forall $v \in Z$ do
    $\texttt{ub}(Z,v) = \texttt{checkIneq}(Z,R,I,v)$;
return $Z$;
```

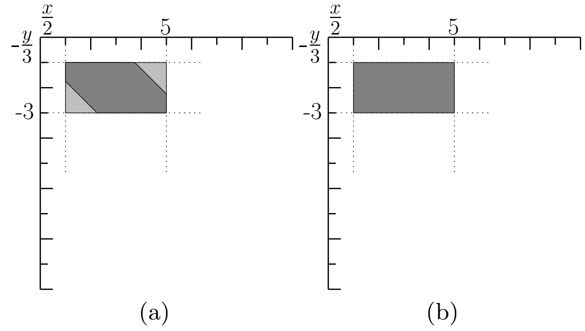
a continuous variable advances with a rate other than one, a variable substitution is performed which has the effect of warping the DBM in the given dimension such that it advances with rate one. For example, in Fig. 4(a) a zone with two continuous variables, x and y, is shown. If x begins increasing at a rate of 2 and y begins increasing with a rate of 3, the warp occurs by substituting x with $\frac{x}{2}$ and y with $\frac{y}{3}$. This has the effect of warping the zone as shown in Fig. 4(b). A DBM can only represent polygons made with 45° and 90° angles. Therefore, the zone in Fig. 4(b) must be conservatively encapsulated in a larger zone which satisfies this requirement. The lighter gray box in Fig. 4(c) shows the encapsulation that includes this zone while using only 45° and 90° angles. The final result of the zone being warped is shown in Fig. 4(d). This example shows how a zone is warped in two dimensions. The general problem of warping the zone for n dimensions can be reduced to warping the zone in two dimensions multiple times. When a rate is negative, the DBM is first warped as described above and this zone is then warped into the negative space. This is accomplished by first swapping the minimum and maximum entries in the zone. In the resulting zone, all 45° angles become 225° angles which cannot be represented in a zone. This can be seen in the darker box shown in Fig. 5(a). To address this problem, the algorithm must encapsulate the zone in a box. The gray box shown in Fig. 5(b) is the result of this encapsulation.



(a) (b)

(c) (d)

Figure 4: Warping a zone in two dimensions.



(a) (b)

Figure 5: Warping a zone by a negative rate.

The algorithm for warping a DBM is shown in Algorithm 6. The first part of the algorithm performs the warping and encapsulation as described earlier. The third loop in Algorithm 6 (line 13) is used when a rate is negative which requires that the values calculated in the previous parts of the algorithm to be warped into the negative space. The resulting DBM Z is recanonicalized and returned. The warp function is shown below:

$$\texttt{warp}(z_1, z_2, r_1, r_2) = r_1 \cdot z_2 - r_1 \cdot z_1 + r_2 \cdot z_1$$

```
Algorithm 6: dbmWarp($Z, R, R'$)
1  forall $\{x,y\} | x \in Z, y \in Z, x \neq y$ do
2      $a = |R(x)/R'(x)|$;
3      $b = |R(y)/R'(y)|$;
4      if $a > b$ then
5          $Z(x,y) = \texttt{warp}(\texttt{ub}(Z,x), Z(x,y), b, a)$;
6          $Z(y,x) = \texttt{warp}(\texttt{nlb}(Z,x), Z(y,x), b, a)$;
7      else
8          $Z(x,y) = \texttt{warp}(\texttt{nlb}(Z,y), Z(y,x), a, b)$;
9          $Z(y,x) = \texttt{warp}(\texttt{ub}(Z,y), Z(x,y), a, b)$;
10 forall $x \in Z$ do
11     $\texttt{nlb}(Z,x) = |R(x)/R'(x)| * \texttt{nlb}(Z,x)$;
12     $\texttt{ub}(Z,x) = |R(x)/R'(x)| * \texttt{ub}(Z,x)$;
13 forall $x \in Z$ do
14     if $R(x)/R'(x) < 0$ then
15         $Z = \texttt{swap}(Z,\texttt{nlb}(Z,x),\texttt{ub}(Z,x))$;
16         forall $y \in Z$ do
17             if $x \neq y \wedge y \neq c_0$ then
18                 $Z(x,y) = Z(y,x) = \infty$;
19 $Z = \texttt{recanonicalize}(Z)$;
20 return $(Z, R')$;
```

5. EXPERIMENTAL RESULTS

A compiler from VHDL-AMS to LHPNs and the state space exploration algorithms for LHPNs described in this paper have been implemented and used to verify the examples in Table 1. Results are compared with the HyTech tool. Our tool verifies VHDL-AMS versions of these examples while HyTech analyzes hybrid automata models.

For the integrator example used in this paper, the property that we wish to verify is that it does not saturate. In other words, *Vout* should never fall below −2V (−2000mV) or rise above 2V (2000mV). This property can be expressed using the following VHDL-AMS **assert** statement:

Table 1: Verification results.

Example	New Method			HyTech
	$\lvert\Psi\rvert$	Time	Verifies?	Time
Integrator (Sat.)	103	0.24	No	0.06
Integrator (No sat.)	142	0.23	Yes	0.08
Diode (Osc.)	115	0.84	Yes	n/a
Diode (No osc.)	132	1.20	No	n/a

```
assert vout'above(-2000) and
  not vout'above(2000)
  report "Integrator saturates"
  severity failure;
```

The **assert** statement has two effects on the LHPN model. First, the **assert** statement is compiled into the LHPN shown in Fig. 6 which fires a transition to set the Boolean signal *fail* to true when the assertion is violated. Second, all enabling conditions in all other transitions are extended to include $\overline{fail}$. The result of this is that when a failure is detected all transitions in the LHPN are disabled resulting in a deadlock. This deadlock is detected and reported as a failure during state space exploration.

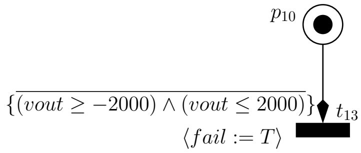

p_{10}
$\{\overline{(vout \geq -2000) \wedge (vout \leq 2000)}\}$
t_{13}
$\langle fail := T\rangle$

Figure 6: An LHPN for the assert statement.

With the addition of this **assert** statement, a deadlock is found during state space exploration of the integrator which means that the circuit can saturate. This can happen, for example, when *Vout* always increases at 22 mV/μs and always decreases at 18 mV/μs. The result is that charge builds up during each cycle until eventually the op amp saturates.

Saturation of the integrator can be prevented using the circuit shown in Fig. 7. In this circuit, a resistor constructed using a switched capacitor is inserted in parallel with the feedback capacitor. With this addition, *Vout* tends to drift back to 0V. In other words, if *Vout* is increasing, it increases faster when it is far below 0V than when it is near or above 0V. To model this affect, the VHDL-AMS is modified to use a range of *Vout* increase of 22 to 24mV/μs when it is below -1000mV, and it uses a range of 16 to 22mV/μs when it is above -1000mV. A similar modification is made for the ranges of rates when *Vout* is decreasing. With these changes, the verification finds that the circuit no longer saturates. While these results can be rapidly found by both tools, the HyTech tool requires the user to translate the VHDL-AMS descriptions by hand into a hybrid automata model.

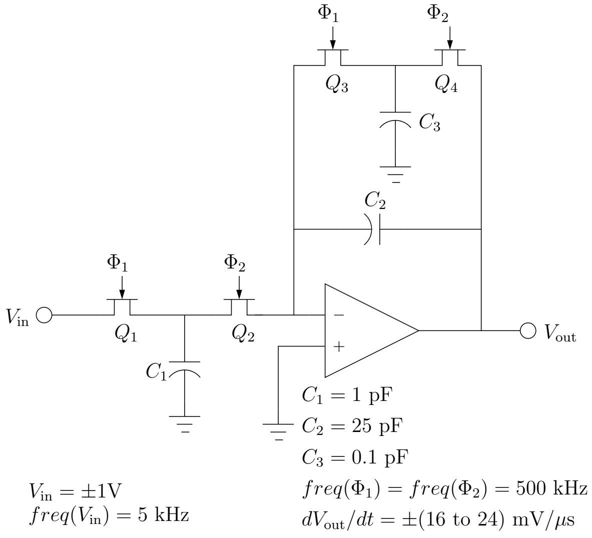


Figure 7: A non-saturating integrator.

The remaining results in Table 1 are versions of the tunnel diode oscillator shown in Fig. 8 [13]. The numerical parameters used for this example are from [11]. The goal of verification is to ensure that $I_{\rm l}$ oscillates for specific circuit parameters and initial conditions. Continuous variables in LHPNs can only change at constant rates. Therefore, to analyze more complicated systems, the continuous operating ranges must be decomposed into regions in which the rate of change is assumed to be constant. This is accomplished using a differential equation discretization method similar to that proposed in [12,13].

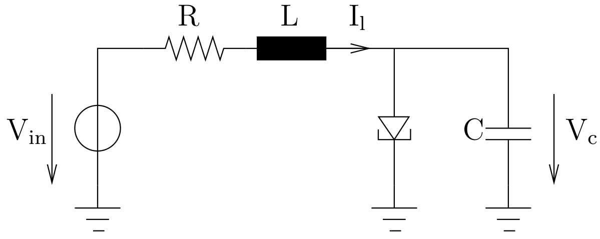


Figure 8: Tunnel diode oscillator circuit ($V_{\rm in}$=0.3v, L=1μH, and C=1pF).

In the model for the tunnel diode oscillator, sixteen discrete regions are required to model the oscillatory and non-oscillatory behavior of the circuit. The property is verified for a range of initial conditions in which $I_{\rm l}$ is between 0.45 to 0.55mA and $V_{\rm c}$ is between 0.4 and 0.47V. As expected, the property verifies with $R = 200\Omega$ in 0.84s after finding 115 state sets, and the property does not verify with $R = 242\Omega$ in 1.20s after finding 132 state sets. We also attempted this verification using the HyTech tool [17], but it is unable to complete due to arithmetic overflow errors. HyTech can complete analysis with less precision on the rates, but the model of the circuit no longer produces oscillation. Therefore, the verification results are incorrect. Our method also outperforms the `PHAVer` model checker on the diode oscillator which verifies it in 72.8s [9]. This demonstrates that our method can provide a significant performance improvement over exact methods without loss in verification accuracy.

Another advantage of the LHPN model over our previously published HPN models is that the systems can be modeled more compactly which results in smaller state spaces. In particular, the HPN diode models presented in [18] require over 2000 zones to represent the state space while the LHPN models require 132 zones or less.

6. CONCLUSION

To gain acceptance of formal verification by AMS designers, it is crucial to allow them to describe circuits using a method they are comfortable with. To this end, this paper describes a method for formally verifying AMS circuits described using a subset of VHDL-AMS. These VHDL-AMS descriptions are compiled into LHPNs which are then analyzed with an efficient zone-based state space exploration algorithm. Zones are extended to represent continuous variables that change at variable rates by using a process known as warping. These warped zones allow for a simpler approximation of the state space while still being able to verify interesting properties of the sample circuits in this paper.

While the results in the paper are promising, there is much research that is still necessary. First, we are developing a SPICE-deck compiler to LHPNs to further improve the ability of AMS designers to use our tool. We are also planning to investigate methods to improve user feedback when a failure is detected or to provide coverage metrics when no failure is found. Finally, we are working on developing complete system models using LHPNs that include both analog and digital hardware as well as embedded software.

7. ACKNOWLEDGMENTS

We would like to thank Goran Frehse of VERIMAG for his help with PHAVer. We would also like to thank Reid Harrison of the University of Utah and Robert Kurshan of Cadence for their comments on this work.

8. 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 R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229. Springer, 1992.

[2] W. Belluomini and C. J. Myers. Timed state space exploration using posets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(5):501–520, May 2000.

[3] W. Belluomini and C. J. Myers. Timed circuit verification using tel structures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(1):129–146, Jan. 2001.

[4] E. M. Clarke and R. P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, June 1996.

[5] 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.

[6] R. David and H. Alla. On hybrid petri nets. *Discrete Event Dynamic Systems: Theory and Applications*, 11(1–2):9–40, Jan. 2001.

[7] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proc. Automatic Verification Methods for Finite-State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1989.

[8] G. Frehse. PHAVer: Algorithmic verification of hybrid systems past hytech. In M. Morari and L. Thiele, editors, *Hybrid Systems: Computation and Control (HSCC)*, volume 3414 of *LNCS*, pages 258–273. Springer, 2005.

[9] 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.

[10] A. Ghosh and R. Vemuri. Formal verification of synthesized analog designs. In *Proc. International Conf. on Computer Design (ICCD)*, pages 40–45. IEEE Computer Society Press, 1999.

[11] S. Gupta, B. H. Krogh, and R. A. Rutenbar. Towards formal verification of analog designs. In *Proc. International Conf. on Computer Aided Design (ICCAD)*, pages 210–217. IEEE Computer Society Press, 2004.

[12] W. Hartong, L. Hedrich, and E. Barke. Model checking algorithms for analog verification. In *Proc. Design Automation Conference (DAC)*, pages 542–547. ACM Press, 2002.

[13] W. Hartong, L. Hedrich, and E. Barke. On discrete modeling and model checking for nonlinear analog systems. In E. Brinksma and K. G. Larsen, editors, *Proc. International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 401–413. Springer, 2002.

[14] L. Hedrich and E. Barke. A formal approach to nonlinear analog circuit verification. In *Proc. International Conf. on Computer Aided Design (ICCAD)*, pages 123–127. IEEE Computer Society Press, Nov. 1995.

[15] L. Hedrich and E. Barke. A formal approach to verification of linear analog circuits with parameter tolerances. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 649–654. IEEE Computer Society Press, Feb. 1998.

[16] S. Hendricx and L. Claesen. A symbolic core approach to the formal verification of integrated mixed-mode applications. In *Proc. European Design and Test Conference*, pages 432–436. IEEE Computer Society Press, 1997.

[17] 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.

[18] S. Little, D. Walter, N. Seegmiller, C. Myers, and T. Yoneda. Verification of analog and mixed-signal circuits using timed hybrid petri nets. In *Automated Technology for Verification and Analysis*, volume 3299 of *LNCS*, pages 426–440. Springer, Nov. 2004.

[19] C. Myers. *Asynchronous Circuit Design*. 2001.

[20] C. J. Myers, R. R. Harrison, D. Walter, N. Seegmiller, and S. Little. The case for analog circuit verification. *Electronic Notes Theoretical Computer Science.*, 153(3):53–63, 2006.

[21] A. Salem. Semi-formal verification of VHDL-AMS descriptions. In *Proc. International Symposium on Circuits and Systems (ISCAS)*, pages 123–127. IEEE Computer Society Press, 2002.

[22] H. Zheng. Specification and compilation of timed systems. Master's thesis, University of Utah, 1998.