Runtime Verification of Nonlinear Analog Circuits Using Incremental Time-Augmented RRT Algorithm

Seyed Nematollah Ahmadyan, Jayanand Asok Kumar, Shobha Vasudevan
Coordinated Science Lab, Electrical and Computer Engineering Department,
University of Illinois at Urbana-Champaign,
{ahmadya2, jasukk2, shohbav}@illinois.edu

***Abstract*—Because of complexity of analog circuits, their verification presents many challenges. We propose a runtime verification algorithm to verify design properties of nonlinear analog circuits. Our algorithm is based on performing exploratory simulations in the state-time space using the *Time-augmented Rapidly Exploring Random Tree (TRRT)* algorithm. The proposed runtime verification methodology consists of i) incremental construction of the TRRT to explore the state-time space and ii) use of an incremental online monitoring algorithm to check whether or not the incremented TRRT satisfies or violates specification properties at each iteration. In comparison to the Monte Carlo simulations, for providing the same state-space coverage, we utilize a logarithmic order of memory and time.**

I. Introduction

Verifying nonlinear analog circuits is a major challenge and an ongoing topic of intensive research. Formal verification methods exhaustively analyze all possible behaviors of the circuit statically. They present a very daunting computational challenge in the domain of analog circuits. A less rigorous, but more practically viable, alternative is runtime verification and monitoring of properties. In runtime verification, a property monitor checks whether a finite set of simulation traces would satisfy or violate a given property specification [13].

Current approaches for runtime verification are inefficient. Runtime verification has two components: the generation of traces and behavior (simulation) and checking and monitoring of properties against the simulated traces. Existing approaches (See [18]) for runtime verification generate transient traces of the system using *Monte Carlo* simulation and monitor properties against these traces.

In this paper, we introduce a technique for runtime verification that is based on the *Rapidly Exploring Random Tree (RRT)* algorithm [10]. Our technique simulates the state-space of a nonlinear analog circuit in a manner different from Monte Carlo simulations. The RRT is a tree data structure that grows rapidly by performing exploratory simulations of system behavior. We use the incremental nature of RRT growth patterns to monitor properties of interest incrementally. Hence, our RRT-based runtime verification framework provides a novel simulation as well as property-checking and monitoring methodology. Previously, RRTs have been extensively used in robotic motion planning [9] and reasoning [8], safety falsification [14][1] and test generation [2][12].

In order to adapt traditional RRTs to runtime monitoring of analog circuits, we introduce the *Time-augmented RRT (TRRT)* algorithm. RRT, being a *tree* data structure, spans the state-space and does not contain loops and cycles. This makes the RRT unsuitable for checking properties like oscillation that need to traverse a cyclic path in the state-space. The RRT explores the entire state-space uniformly without any bias towards a particular dimension. Thus the growth along the time dimension might be very small for a large state-space with many dimensions. We address these issues in TRRT data structure. We augment the state-space of an analog circuit with the time dimension, providing a state-time space for the time-augmented RRT to grow. The state-time space adds the time dimension to the $n-$dimensional state-space vector. In order to ensure forward progress in time, we introduce a sampling bias in the time dimension, without violating the probabilistic completeness property of the time-augmented RRT.

Our runtime verification technique based on TRRTs works as follows. Design specification properties are provided to describe the temporal and logical behavioral model of the analog signals. Given initial state(s) and input parameters of a system (including uncertainty and variation parameters), our algorithm randomly samples the state-time space of the circuit and expands the TRRT toward the new samples through simulation. The TRRT is constructed incrementally starting from the specified initial state. At every incremental iteration, an edge corresponding to a single simulation trace from the previous (initial) state to the next (final) state is added to the tree. As a result, the constructed TRRT consists of many randomized simulation traces as edges. At every iteration, our monitoring algorithm checks the newly added edge against given properties for violation. The properties we check include both analog properties, meaning input/output properties that do not involve previous state information, and temporal properties, i.e., properties that are stateful. We use the STL/PSL properties [13], with a few modifications for TRRT-based checking. We define the operator *Norm* for computing distance the between vectors as well as the jitter property in a manner that is not amendable to Monte Carlo simulation, but is verifiable with TRRTs.

Our technique has many benefits over state-of-the-art approaches for runtime verification. First, ours is more efficient. A major source of inefficiency in Monte Carlo simulations is the overlapping of simulation traces. For a given circuit, a majority of the simulation traces browse the same path in the state-space during runtime. Therefore, they share the same path and overlap with each other. Repeated simulation of the same path in the circuit does not provide any new information, and result in poor performance. However, the TRRT-based method incrementally grows the TRRT in the state-time space. TRRT is persistently aware of the state-space all the time. TRRT, being a tree data structure, always grows toward unique samples in the state-space such that two traces never coincide. In TRRT the direction of the growth is always toward a state as yet unexplored and every simulation trace is unique. At every iteration, the tree traverses and covers more of the state-space. Consequently, TRRT does not allow repeated sampling of the same sequence of nodes and prohibits overlapping of traces in the state-space. Since we conceptually arrange Monte Carlo's linear simulation traces in a tree data structure, we have an average logarithmic efficiency in simulation performance and memory to achieve the same state-space coverage as Monte Carlo.

Second, our monitoring algorithm proceeds incrementally. For most cases, we only have to check the incremented edge to the TRRT to decide whether a property has been violated. Thus, our incremental monitoring algorithm using TRRT is more efficient than monitoring the entire trace [8].

Finally, by using TRRTs, we efficiently verify properties that require a comparison between entire trace. Such properties include jitter and deviation. Monte Carlo simulations, as they simulate only one trace at a time and possess no knowledge of the state-space, are not well suited to checking those properties. On the other hand, the TRRT can maintain information and verify multiple traces simultaneously because of its step-by-step growing data structure.

We model circuit states and inputs as continuous finite variables without discretizing them. We use SPICE to simulate circuit behavior. Our methodology accurately models the continuous-time behavior of analog circuits.

Our main contributions are as follows.

- We propose an incremental property checking and runtime monitoring algorithm for nonlinear analog circuits that utilizes TRRT to verify design specification properties. Our algorithm is incremental in nature and more efficient than previous strategies for runtime verification.
- We introduce a time-augmented rapidly-exploring random tree (TRRT) algorithm. TRRT has an augmented time dimension and a biased sampling algorithm in that dimension. TRRT provides the same coverage as Monte Carlo, while utilizes the logarithmic order memory and time. TRRT prohibits simulation trace overlap.
- We define the semantics for our property specification language.

To demonstrate the effectiveness of the proposed approach, we

978-3-9815370-0-0/DATE13/

applied our technique in several case studies. We first used our methodology on a tunnel diode circuit as a proof of concept. Then we showed the scalability and practicality of our technique by using it to verify a PLL circuit.

Rest of this paper is organized as follows. Preliminaries are presented in Section II. In Section III we define the syntax and semantics of our property specification language. We describe our runtime monitoring algorithm in Section IV. We conclude in Section VI.

II. Preliminaries and background

A. Models for nonlinear analog circuits

Modern circuit simulators such as SPICE, use ordinary differential equations to model nonlinear analog circuits. It is possible to extract those equations by applying modified nodal analysis (MNA) to the circuit netlist. Therefore, the transient response of an analog circuit is modeled as:

$$f(\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t)) = 0 \tag{1}$$

where $t \in [0, \infty)$ and f is a nonlinear function. Let $\mathbb{S} \subseteq \mathbb{R}^n$ denote the continuous state-space of the circuit. Let $\mathbb{U} \subseteq \mathbb{R}^m$ denote the input space of the circuit. $\mathbf{x}$ denotes the state variables, and $\mathbf{u}$ denotes the input variables of the circuit. $\mathbf{x}(t)$ denotes the state of the circuit at time t. The initial state of the circuit is $\mathbf{x}(0)$.

The circuit's trace in the time interval $[t_1, t_2]$ is the path taken by the circuit from state $\mathbf{x}(t_1)$ to state $\mathbf{x}(t_2)$. Given the circuit's input at time interval $[t_1, t_2]$ and state of the circuit at time t_0, SPICE can compute the circuit's trace by simulating the circuit using the following equation.

$$\mathbf{x}(t) = \mathbf{x}(t_1) + \int_{t_1}^{t} f(\mathbf{x}(t'), \mathbf{u}(t'))\Delta t' \tag{2}$$

B. Rapidly-exploring Random Trees

Our algorithm is based on the *Rapidly-exploring Random Tree (RRT)* algorithm [9]. The RRT algorithm is basically a tree data structure designed to explore as rapidly state space as possible. The root of the tree is the initial state in the state space $\mathbb{S}$. The tree is incrementally *grown* by adding an edge between an existing node and a new point (*i.e.*, state) selected from the state space. The selection of these new points determines the manner in which the tree grows in the state space. Typically, these new points are selected at random by uniformly sampling the state space.

Let $\mathbb{G}$ be the RRT data structure. Each node of $\mathbb{G}$ corresponds to a state in $\mathbb{S}$, *i.e.* a unique set of values assigned to the state variables $\mathbf{x}$. Each edge represent a trajectory (Equation 2) of the system from initial condition $\mathbf{x}$ for a given assignment of values to the input variables $\mathbf{u}$. For every new generated state q_{sample}, the RRT algorithm will find a closest state, q_{near}, and will determine which trajectory (Equation 1) for any $u \in U$ will brings node q_{near} closer to the sampled state. The closest state is determined based on Euclidean distance between two states. RRT expands from q_{near} towards q_{new}. Figure 1 shows the growth of the RRT tree toward a given sample node. The RRT rapidly visits unexplored regions of the state space [9] and as the number of samples approaches infinity, the RRT provably covers the entire state space [9].

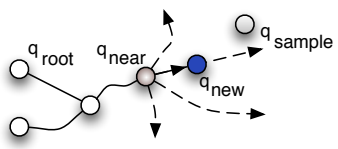


Fig. 1: Growth of RRT by adding a new node sampled from the state space.

III. Analog Property Specification

We use a property specification language based on STL/PSL to define our properties [13][7][11]. We use a subset of the operators defined in STL/PSL, and define a few of our own to suit the RRT verification framework. As in PSL, the analog layer is used to describe the properties of continuous variables and vectors, and temporal layer is used to reason about the temporal behavior of the circuit.

We define the **Norm** operator for both the analog and temporal layers. We also express the **jitter** property in a manner that is conducive to RRT-based verification. We describe the syntax and semantics of all the operators we use in both layers. The rest of this section describes the syntax and semantics of the analog and temporal layers of properties.

A. Syntax

1) Syntax of the analog layer: The grammar for the analog syntax is as follows.

$$\phi ::= \texttt{var} | \texttt{const} | f(\phi, \ldots, \phi) \tag{3}$$

var is a continuous finite variable, and can be a single-dimensional vector, much as x_i denotes to a single waveform in simulation, or can be any subset of the circuit's state vector, like $< x_{i_1}, x_{i_2}, \ldots, x_{i_{n'}} >$, where the index set $\{i_1, \ldots, i_{n'}\}$ is specified by the user. *const* is a finite constant. Finally, f can be any of the following functions:

- Shift
- Binary operators, including $\{+, -, \times, /\}$[1]
- Norm

The semantics of the above functions are described in Section III-B1.

2) Syntax of the temporal layer: We define the temporal layer to reason about time. Let ϕ be an atomic proposition. For every state (sub)vector $\mathbf{x}$, we associate a time instance of the form $\mathbf{x}(t)$ where $t \in \mathbb{T}$. Set $\mathbb{T}$ is the set of all possible times, and it is defined as $T := \{x | x = k \times \Delta t, k \in \mathbb{Z}^+\}$ where Δt is the minimum discrete time resolution. The time interval is an array of $\mathbf{x}(t)$ with a variable t. Time intervals can be fixed, like $t \in [30, 40]$, or can be relative, like $t \in [t, t + 300]$. In both cases, we write them as $t \in [t_i, t_j]$. Moreover since we monitor the behavior of the system for a finite time interval, temporal modalities are bounded to intervals of the form $[i, j]$, where $0 < i < j \leq T_{max}$, and $i, j \in \mathbb{T}$ where $T_{max} = \sup(\mathbb{T}) = k_{max} \times \Delta t$.

Similar to [13], we define the temporal layer as follows.

$$\begin{aligned} \varphi \quad &= \quad p \mid \phi[a:b] \ \star \ \phi[a:b] \mid \texttt{not}\ \varphi \mid \varphi \bullet \varphi \\ &\mid \quad \texttt{eventually}[a:b]\ \varphi \mid \varphi\ \texttt{until}[a:b]\ \varphi \mid \mathcal{J}[a:b](\varphi) \end{aligned}$$

v is a propositional variable; the comparison operator $\star$ includes $\{\leq, <, \geq, >, \doteq, \not\doteq\}$. The logical operator $\bullet$ includes logical and, or, xor, xnor, nand and nor. The semantics of the above functions and operators are defined in Section III-B2.

We use the notation ϕ for analog and φ for temporal formulas.

B. Semantics

1) Semantics of analog layer: The semantics of the analog layer are defined as the function of f over the state (sub)vector ϕ. f can be any of the following functions:

- Shift: Shift is defined as changing the index of time dimension of a variable along the same trace on the simulation, i.e., $\texttt{shift}(\phi, \texttt{const})[t] = \phi(t + \texttt{const})$.
- Binary function[2] f: $f(\phi_1, \phi_2)[t] = f(\phi_1[t], \phi_2[t])$.
- **Norm**(ϕ, p), Norm(ϕ_1, ϕ_2): returns the p-norm of ϕ, or the L^2-norm for computing the distance between ϕ_1 and ϕ_2, assuming both propositions have the same dimension. Norm is used to measure the distance of the state-vector against another vector or a constant. L^2-norm is defined as $\texttt{Norm}(x, y) := ||x - y|| = \sqrt{\sum_{j=1}^{n'}(x_{i_j} - y_{i_j})^2}$, and the p-norm, assuming $p \geq 1$, is defined as $\texttt{Norm}(x, p) := ||x||_p = (\sum_{j=1}^{n'} |x_{i_j}|^p)^{\frac{1}{p}}$.

2) Semantics of the temporal layer: For the temporal layer, we mostly use the same semantics proposed in [13], with some modifications as follows. Comparison operator $\star$ includes $\{\leq, <, \geq, >, \doteq, \not\doteq\}$. To reason about equivalence in the analog domain, we use the notation $x(t) \doteq y(t')$ to indicate that $||x(t) - y(t')|| < \epsilon$. Equivalence operator is satisfied if and only if for any given $t \in [t_i, t_j]$, state $x(t)$ remains within ϵ-envelope around $y(t')$ for any given $t' \in [t'_i, t'_j]$. The following definition easily expands to $\leq, \geq$, and $\not\doteq$.

[1] With the exception that $\frac{\phi_1}{\phi_2}$ is well-defined if and only if $0 \notin \phi_2$.

[2] For simplicity, we use the notation $\diamond$ for the binary function f. Therefore, $(\phi_1 \diamond \phi_2)[t] = \phi_1[t] \diamond \phi_2[t]$.

The logical operator $\bullet$ includes logical and, or, xor, xnor, nand & nor. The semantics of `not` are defined as $\mathbb{M} \models (\texttt{not}\varphi)$ `iff` $\mathbb{M} \not\models \varphi$. Similarly, $\mathbb{M} \models (\varphi_1$ `and` $\varphi_2)$ `iff` $\mathbb{M} \models \varphi_1$ `and` $\mathbb{M} \models \varphi_2$ and so on. $\mathbb{M}$ is the circuit's simulation abstraction provided by the RRT $\mathbb{G}$.

The semantics of *Until* are defined in [13]. The *Eventually* operator can be defined using *Until* as follows. The temporal modalities $\Diamond$ (*eventually*, sometimes in the future) and $\Box$ (*always*, from now on forever) are derived as follows: $\Diamond\varphi \equiv true$ `until` φ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$. By combining the above, we obtain the *infinitely often* $\varphi \equiv \Box\Diamond\varphi$ and the *eventually forever* $\varphi \equiv \Diamond\Box\varphi$.

Jitter is an undesired deviation from true periodicity of an assumed periodic original signal. The basic jitter operator ($\mathcal{J}(x,t)$) will compute the deviation in time using
$\mathcal{J}(f,x,t) = \max_{0\leq t\leq T_{max}}(x(t)-x(t-f)) - \min_{0\leq t\leq T_{max}}(x(t)-x(t-f))$.

For periodic signals, the above definition is equivalent to a maximum deviation of state vector $x(t)$ from other state vectors at the same period (including other states at time $t, t-f, t-2f, \ldots$). We use that idea to create a recursive $\mathcal{J}$ operator model for computing jitter in RRT later in Section IV-B. We also extend the jitter operator to compute the deviation of non-periodic signals.

IV. TRRT-based Runtime Verification Algorithm

Our goal is to verify that an analog circuit $\mathbb{M}$ satisfies a property Φ. A summary of the steps (Figure 2) in our TRRT-based runtime verification algorithm is as follows.

1) We construct a TRRT $\mathbb{G}$ (Section II.B) to represent a set of feasible traces of the analog circuit $\mathbb{M}$. We construct $\mathbb{G}$ by initializing it to a known operating point of the circuit and then growing it step by step.
2) In each step, we employ a *monitor* to check whether the property Φ is violated by any of the traces represented by the TRRT $\mathbb{G}$.
3) If the monitor does not detect a violation, we grow the TRRT by one more step. We repeat this process iteratively until a violation is detected or a user-specified limit on the number of steps is reached.

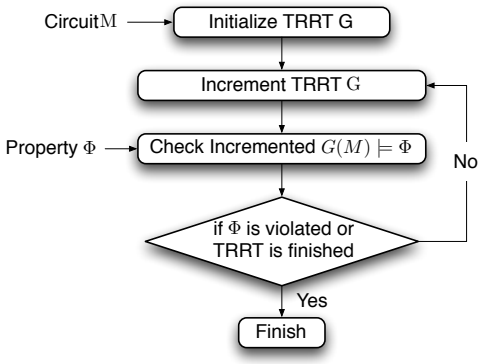


Fig. 2: Flowchart of TRRT-based runtime monitoring algorithm

In each step of our algorithm, we grow the TRRT $\mathbb{G}$ by adding a single edge to the tree (Section II.B). Each edge in $\mathbb{G}$ corresponds to a single transient circuit simulation of length Δt. For several types of properties, our monitor can detect a violation by checking only the newly added edge of the TRRT. Therefore, for those types of properties, our algorithm is highly efficient since it can perform verification in an incremental manner.

With each step, the TRRT grows. TRRT will quickly explore and provide a high coverage of the reachable state-time space of the circuit $\mathbb{M}$ [2]. Therefore, we can have more confidence in the verification results that we obtain using our algorithm than the verification results obtained from random Monte Carlo simulations. State-time space is the circuit's state-space augmented with a dimension corresponding to time.

Algorithm 1 shows our TRRT-based runtime monitoring algorithm. The inputs to our algorithm are i) the circuit $\mathbb{M}$, ii) the initial state of $\mathbb{M}$, and iii) the properties φ to be verified on $\mathbb{M}$.

We now describe the steps of our algorithm in detail.

Algorithm 1 TRRT-based runtime monitoring algorithm

```
Data: circuit M, initial state x(0), properties φ
G = InitializeTRRT(x(0)) ;
InitializeMonitor(φ) ;
for i ← 1 to K do
    q_sample = UniformSampling(S) ;
    q_sample[n] = rand([i/K × rand(0, T_max)], T_max) ;
    q_near = FindNearestNodeInTree(S, q_sample) ;
    u(t) = GenerateNewInput(S) ;
    q_new = Simulate(M, q_near, u(t), Δt) ;
    G.expand(q_new);
    Monitor(G, q_new, φ) ;
    if G ⊭ φ then
        return violation ;
    end
end
```

A. Constructing the TRRT for the circuit

This section describes how we extended the TRRT algorithm for runtime verification of analog circuits.

In real-world circuits, a state might be revisited during circuit operation. Oscillation circuits are very good examples of that scenario. Traditional TRRTs are tree data structures that span the state-space, and therefore do not contain any cycles. Such TRRTs cannot be used for verifying oscillation properties. In order to address that issue, we modify TRRTs to include a notion of time as well.

For a circuit with n state variables, we construct a TRRT $\mathbb{G}$ of $n+1$ dimensions corresponding to the state-time space of the circuit. State-time space is the n-dimensional state-space augmented with a dimension corresponding to time. Each node in the tree $\mathbb{G}$ is an $n+1$-dimensional vector denoted by $< x_1, x_2, \ldots, x_n, t >$ that corresponds to an n-dimensional state vector $< x_1, x_2, \ldots, x_n >$ and a time variable t. Let q be a point in the state-time space of circuit $\mathbb{M}$. We use the notation $q[x]$ to indicate the state vector and $q[t]$ to indicate the time variable of q.

Each edge in the TRRT $\mathbb{G}$ denotes a transient simulation of the circuit. We annotate each edge of $\mathbb{G}$ with the simulation time stamp t and the inputs to the circuit at that time $u(t)$. We use a discrete time step Δt for simulating each edge of the TRRT. Therefore, in the process of constructing the TRRT, we discretize the behavior of the circuit. We choose Δt to be small enough such that we do not discard the relevant behavior of the circuit. The minimum time step Δt should satisfy the *Nyquist* criterion $\frac{1}{\Delta t} \geq 2f_{\texttt{max}}$, where $f_{\texttt{max}}$ is the maximum operating frequency of the circuit [3].

To build the TRRT $\mathbb{G}$, we first select an initial state as the root of the tree $\mathbb{G}$. Our algorithm performs standard DC operating point analysis over M and sets the computed operating point as the initial state of the circuit. Alternatively, we allow the user to specify the initial state of the circuit.

In our algorithm, we grow the TRRT in a step-by-step manner. In each step, we obtain a point $q_{\texttt{sample}}$ by randomly sampling the state-time space of $\mathbb{M}$. We then grow the TRRT towards the point $q_{\texttt{sample}}$ as follows. We first find the closest node (in terms of Euclidean distance) to $q_{\texttt{sample}}$ in the TRRT $\mathbb{G}$, namely $q_{\texttt{near}}$. As in the classical TRRT algorithm (Section II.B), we determine the best possible trace from $q_{\texttt{near}}$ toward $q_{\texttt{sample}}$ and generate an input $u(t)$ to the circuit to follow that trace. We simulate the circuit $\mathbb{M}$ from state $q_{\texttt{near}}$ for simulation time Δt using input $u(t)$, and determine the resulting state $q_{\texttt{new}}$. We then add $q_{\texttt{new}}$ as the new reached state to the TRRT $\mathbb{G}$.

The transient circuit simulation always progresses in time. We wish to model that in the TRRT growth as well. In order to achieve that while growing the TRRT, we filter out the candidates for the closest node that have a time annotation higher than that of $q_{\texttt{sample}}$. In other words, $q_{\texttt{sample}}[t] \geq q_{\texttt{near}}[t]$. Therefore, we ensure that the time notation of the parent node in $\mathbb{G}$ is always smaller than that of its children. As a result, our TRRT correctly models the progression of time in simulation traces of the circuit.

The classical TRRT algorithm tries to explore the entire space uniformly with no bias towards any particular dimension. Therefore, if there are many circuit state variables, the TRRT's growth in the time dimension may be very small. Consequently, it may take a large number of growth steps before the TRRT contains long simulation traces. Therefore, we modify the classical TRRT algorithm to improve

the efficiency of our methodology. We modify the classical TRRT algorithm by introducing a small bias towards the time dimension. We ensure that the bias does not alter the probabilistic completeness property of the TRRT.

Let $T_{\texttt{max}}$ be the maximum simulation time specified by the user. We bias our random number generator by adding a probabilistic offset to the time variable. The default random generator for $q_{\texttt{sample}}$ is $q_{\texttt{sample}}[t] = \texttt{rand}(0, T_{\texttt{max}})$. Function $\texttt{rand}(a, b)$ uniformly samples a number in the interval (a, b). We wish to shift the time bias as i, the number of iterations, increases. Therefore, we use the following probabilistic offset to bias the time: $\frac{i}{k} \times \texttt{rand}(0, T_{\texttt{max}})$. That bias does not violate the probabilistic completeness property of TRRTs, since $\lim_{K\to\infty} \frac{i}{k} \times \texttt{rand}(0, T_{\texttt{max}}) = 0$.

With our bias, we calculate the time index of each new sample as

$$q_{\texttt{sample}}[t] = \texttt{rand}([\frac{i}{k} \times \texttt{rand}(0, T_{\texttt{max}})], T_{\texttt{max}}) \tag{4}$$

where k is the maximum number of iterations and i is the current iteration of the algorithm. The t index in each node corresponds to the time variable.

We use the notation $\mathbb{G}_i$ to indicate the TRRT $\mathbb{G}$ at the i^{th} iteration of the algorithm. After adding a new edge to the $\mathbb{G}_{i-1}$, the monitoring algorithm *Monitor* checks the incremented tree $\mathbb{G}_i$ against the property Φ.

B. TRRT-based incremental monitoring algorithm

The monitoring algorithm *Monitor* first parses the analog property Φ (Section III) into a parser tree $\mathbb{P}$. The parser breaks down the formula into smaller sub-formulas. Each sub-formula can be an analog or temporal formula as described in Section III. The parser performs that procedure recursively until all the sub-formulas are *atomic propositions*. An atomic proposition is a formula that is either *true* or *false* and cannot be broken down into simpler sub-formulas. Every leaf in $\mathbb{P}$ corresponds to an atomic proposition.

Monitor starts by checking the atomic propositions in the leaves of the parser tree $\mathbb{P}$. For every atomic proposition φ, the monitoring algorithm marks every node q in TRRT $\mathbb{G}$ such that $q \models \varphi$. Algorithms 2 and 3 describe how *Monitor* checks analog and temporal properties, respectively. The algorithm then moves from the leaves of $\mathbb{P}$ upwards to the top formula (i.e., the root of $\mathbb{P}$), checking every sub-formula stored in $\mathbb{P}$. *Monitor* terminates when the root of $\mathbb{P}$ is reached.

If the atomic proposition is an analog formula, *Monitor* employs Algorithm 2 to evaluate it. The evaluation of an analog formula involves computations using scalar data. These computations do not involve sequences in time. The algorithm marks every node in $\mathbb{G}$ in which the proposition evaluates to *true*.

The shift function can be implemented by traversing the TRRT $\mathbb{G}$ backward in a trace from a leaf toward the root of the tree. The TRRT consists of a set of simulation traces that are continuous in time. Therefore, a path from any node to the root of the tree is a complete *reversed* simulation trace of the circuit. Hence, traversing the tree backward through each node's parents is the same as moving backward in simulation. Similarly, the *maxSibling* and *minSibling* functions, which we use later in jitter computation, traverse the TRRT among siblings of each node (instead of parents) and would return a sibling with the maximum or minimum value. The basic operators and norm functions are computed according to the semantics in Section III-B. Finally, the algorithm moves on to the next leaf in the parser tree.

Since most of analog and temporal properties are associated with a single node, by adding a new node, we don't have to check the entire TRRT $\mathbb{G}$ to verify those properties. In most cases, verification of satisfaction and violation can be deduced by only checking the last node q_{new} against the incremented tree. As a result, because of the iterative construction of TRRT algorithm 2 can be performed at $O(1)$ for most operators. The shift operator is an exception with the worst-case complexity of traversing the tree from a leaf to the node, which is $O(\log n)$.

In order to verify a formula with temporal operators, *Monitor* employs Algorithm 3. This algorithm analyze sequences of nodes in $\mathbb{G}$, each of which corresponds to a transient simulation in the circuit. An interval in which the proposition is defined is specified in the proposition. At every iteration, the algorithm checks whether $\mathbb{G}$ satisfies or violates the temporal formula for traces that lie within that interval.

Algorithm 2 Analog checking algorithm *Monitor*($\mathbb{G}, q_{\texttt{new}}, \phi$)

```
Data: Analog Formula φ, TRRT 𝔾, new node q_new
switch φ do
    case const
        return const ;
    case f(φ_1, ..., φ_n)
        for i ← 1 to n do
            Check(φ_i) ;
        end
        switch f do
            case Shift
                q_parent = q_new;
                while Parent(q_parent)[t] ≠ φ_2 do
                    q_parent = Parent(q_parent) ;
                end
            case Binary Operator ◇
                if f is division function then
                    Check φ_2 ∉ (0 − ϵ, 0 + ϵ) ;
                end
            case Norm
                Compute L_2 or L_p norm ||φ_1||_φ2 or ||φ_1 − φ_2|| ;
            case Max(Min)-Sibling
                foreach Node q_sibling in Child(Parent(q_new)) do
                    Max = Max(q_sibling) ;
                    Min = Min(q_sibling) ;
                end
        end
        return f(φ_1, ..., φ_n) ;
    if 𝔾 ⊨ φ then
        mark q_new ;
    end
end
```

An example would be a decision on whether $\mathbb{G} \models (x[t, t + 100] < y[0, 100])$.

In order to incrementally decide whether $q_{\texttt{new}} \models \texttt{Eventually}\ \varphi$, we add a Boolean variable `IsEventuallySatisfied` to each node in TRRT $\mathbb{G}$. `IsEventuallySatisfied` is true, if and only if at least one node along the path from $q_{\texttt{new}}$ along its parents to the root of $\mathbb{G}$ satisfies φ (honoring the time interval [a,b], on the path and filtering out other nodes). Algorithm 3 shows how we compute and update **Eventually** operator on TRRT.

To incrementally decide whether $q_{\texttt{new}} \models \varphi_1\ \texttt{until}\ \varphi_2$, we add two additional variables to each node in TRRT. The first Boolean variable is $\texttt{always}\varphi_1\texttt{TillNow}$ which indicates that along the path from $q_{\texttt{new}}$ to its parents in the interval $[a, b]$, the proposition φ_1 is always satisfied. The other variable is $\texttt{NumberOf}\varphi_2\texttt{TillNow} \in \{0, 1, \texttt{many}\}$, which counts the number of nodes that satisfy proposition φ_2. In **Until** proposition, violation occurs when φ_1 does not hold until φ_2. Our method for finding the violation is sketched in Algorithm 3.

The TRRT algorithm incrementally builds the tree by adding simulation traces edge by edge. As a result, for the majority of formulas in our semantics, checking only the new edge is enough to verify or find a violation of the formula over $\mathbb{G}$. However, for some temporal properties we may have to go back in time or more concisely traverse the TRRT $\mathbb{G}$ from the new leaf node upward, through its parents, towards the root of the tree until we can determine the status of the property. In the worst-case, that can take $O(\log n)$, where n is the size of TRRT $\mathbb{G}$. Since the size of the tree is finite, and by definition of the tree, there is no loop in the tree, our algorithm always terminates.

V. Experimental Results and Discussion

To evaluate our approach, we have implemented our algorithm in the C++ language. For simulating the circuit, we used Synopsys HSPICE and developed the interface between HSPICE and our tool.

We show the results for two nonlinear systems. The first case study involves a tunnel diode oscillator that we used as a proof of concept. The second case study is a Phase Locked Loop (PLL) circuit.

1) Tunnel diode oscillator: To illustrate our methodology, we considered a tunnel diode oscillator that is a well-known nonlinear analog circuit. The resonant tunneling of the tunnel diode allows the current

Algorithm 3 Temporal checking algorithm *Monitor*($\mathbb{G}, q_{\texttt{new}}, \varphi$)

Data: Temporal Formula φ, TRRT $\mathbb{G}$, new node $q_{\texttt{new}}$
switch φ **do**
  case v
    return v ;
  case $\phi_1 \star \phi_2$
    check if $q_{new} \models \phi_1 \star \phi_2$ // Analog properties
  case $\varphi_1 \bullet \varphi_2$
    check if $q_{new} \models \varphi_1 \bullet \varphi_2$ // Temporal properties
  case *Eventually* φ*[a,b]*
    if $q_{\texttt{new}} \models \varphi$ *or Parent(*$q_{\texttt{new}}$*)*.IsφSatisfied **then**
      $q_{\texttt{new}}$.IsφSatisfied = true ;
    end
  case φ_1 *until* φ_2
    if $q_{\texttt{new}} \models \varphi_2$ **then**
      $q_{\texttt{new}}$.NumberOfφ_2TillNow =Parent($q_{\texttt{new}}$).NumberOfφ_2TillNow + 1
    end
    if $q_{\texttt{new}} \models \varphi_1$ *and Parent(*$q_{\texttt{new}}$*)*.Alwaysφ_1TillNow **then**
      $q_{\texttt{new}}$.Alwaysφ_1TillNow = true
    end
    if $q_{\texttt{new}}$.NumberOfφ_2TillNow $= 1$ *and Parent(*$q_{\texttt{new}}$*)*.Alwaysφ_1TillNow=*false* **then**
      return violation ;
    end
  case $\mathcal{J}$
    for *Every child node* v_i *in Shift(*$q_{\texttt{new}}$*, t)* **do**
      $\mathcal{J}(q_{\texttt{new}})$ = max($\mathcal{J}(v_i)$)-min($\mathcal{J}(v_i)$) ;
    end
  if $\mathbb{G} \models \varphi$ **then**
    mark $q_{\texttt{new}}$;
  end
end

to decrease as voltage increases for some range of voltages. We used the circuit shown in Figure 3.

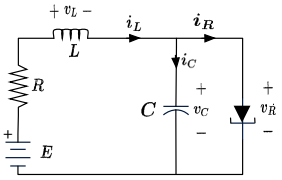


Fig. 3: Tunnel-diode oscillator circuit

This circuit has a two-dimensional state-space. The state equations are modeled as

$$\dot{i_L} = \frac{1}{L}(v_{in} - R \times i_L - v_C) \quad (5)$$

$$\dot{v_c} = \frac{1}{C}(-i_d(v_c) + i_L) \quad (6)$$

$i_d(v_c)$ describes the nonlinear tunnel diode behavior.

We wished to verify whether or not for a given variation in the voltage source and uncertainty parameters in tunnel-diode models and for given initial conditions, the circuit satisfies the following oscillation property. We modeled the voltage source variation as $V_{in} = V_0 + p_1$, where $V_0 = 300mv$ and the tunnel diode uncertainty was modeled as $i_d(x) = x^3 - 1.5x^2 + 0.6x + p_2$. We assumed that the distribution of both variation parameters (p_1 and p_2) was uniform.

The oscillation property under consideration is as follows [6]. For oscillation, the current i_L should cycle between 0.02 and 0.06 indefinitely. Within the time interval $[0, 1us]$, infinitely often whenever the $Norm(i_L)$ reaches 0.02, it will reach this value again within the time interval $[0, 6e-7]$. Also, the same property applies for i_L with amplitude 0.06. Formally $\forall\Box[0:1us](\forall\Diamond[0:0.6us](i_L \leq 0.02)) \wedge \forall\Box[0:1us](\forall\Diamond[0:0.6us](i_L \geq 0.06))$

Figure 4 shows the results of the tunnel-diode analysis using TRRT. Figure 4a shows the state-time space of the tunnel diode circuit with the voltage source variation modeled by $p_1 \in [-0.05mv, 0.05mv]$ and $p_2 \in [-0.005, 0.005]$. The TRRT time resolution was set to $\Delta t = 10ns$, and we executed the algorithm for 20,000 iterations. Figure 4b shows the projection of the state-time space into the time dimension, which is the state-space of the circuit. As shown in figure 4b, for many simulation traces, the circuit oscillate fully; however, for some branches of the TRRT, the oscillation was limited and did not meet the specification. Those branches failed to satisfy the design constraints for oscillation. Thus, this circuit is not verified. Figure 4c Shows the same circuit for the same initial condition, but with reduced variation parameters $p_1 \in [-0.005mv, 0.005mv]$ and $p_2 \in [-0.0005, 0.0005]$. We didn't find any violation of the specification in this circuit.This example shows that even for a common initial state, the bound on variation parameters can lead to the violation of design specifications. The final experiment, shown in Figure 4d, was the tunnel diode example. The parameters in the tunnel diode model was set up by the same bounded variation as shown in Figure 4c, but with a different initial state. In this case, the circuit would not oscillate at all.

2) Phased-Locked Loop circuit: PLL is a circuit that generates an output signal whose phase is related to the phase of an input reference signal. PLL circuit typically consists of a reference signal generator, a voltage-controlled oscillator (VCO), a phase-frequency detector (PFD), a loop filter, and a feedback loop.

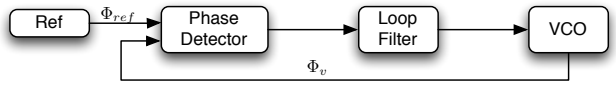


Fig. 5: Phase-Locked Loop (PLL) circuit

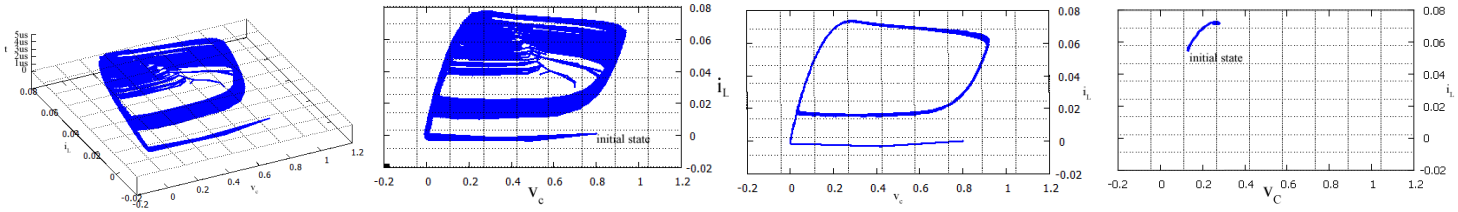
Figure 5 shows the basic architecture of a PLL. In our simulation, we set the initial condition in the unlocked state of the PLL. When the PLL is out of lock, the frequencies of the input and output signals are different. The filter suppress the higher harmonics. Consequently, there will be a DC component that will pull the average output frequency of the VCO up or down until the PLL locks. When the output of the filter is stable, PLL has locked to the input signal. We used a PLL circuit similar to the one described in [5] with $\Phi_{ref} = 1$MHz and $f_0 = 1.01$MHz. The input signal was generated through a fixed sinusoidal voltage source. HSPICE simulation was performed at the highest run level for maximum accuracy. To verify the uncertainty parameters in PLL, we added variation parameters to sources at the phase-detection block at each iteration. The variations were uniformly generated from interval $[-0.001, 0.001]$. We executed the TRRT for 30,000 iterations. Our PLL circuit had 17 states.

We used the output signal of the loop filter to verify the locking of the PLL. When PLL locks, the output signal eventually becomes stable, and there is no more deviation in the signal, Except for some small deviations due to the phase-detector operations. We define the `PLL-locking-property` as eventually forever the jitter on the analog signal $Norm(v_1 - v_2)$ becomes less than $50mv$ in $2us$ interval where v_1 and v_2 are outputs of the loop filter block. Therefore eventually forever jitter on $v_1 - v_2$ is smaller than $50mv$. That property means that the deviation of the given signal has to become less than 0.05 in a $2us$ interval, so that it can be considered stable. Thus, we can assume that PLL has locked.

Figure 6 shows the deviation trace of the norm distance between the loop filter's output generated by TRRT under uncertainty parameters in the phase-detector block. Our algorithm founds no violation in the pll-locking-property, so we assume the output of the loop filter eventually becomes stable forever.

A. Comparison of our algorithm to standard Monte Carlo simulation

To simulate n nodes for $m\Delta t$ time, Monte Carlo requires mn simulation samples. On the other hand, to simulate the circuit for the same time, assuming that the TRRT will be fully populated with output degree of d for each node, the TRRT algorithm would simulate $d^{m-1} - 2$ nodes. That is equivalent to $\frac{d^{m-1}-2}{m}$ Monte Carlo simulation samples (the number of the edges in the fully populated TRRT over time). Therefore, TRRT is more efficient than the standard Monte Carlo algorithm. As a result, to achieve the same state-space coverage as Monte Carlo, we have an amortized logarithmic efficiency in simulation performance and memory, since we are conceptually arranging Monte



(a) State-time space of TRRT after oscillation with uncertainty

(b) Projection of TRRT's state-time space into state-space for oscillation with uncertainty

(c) TRRT's state-space projection for oscillation of a verified circuit

(d) TRRT's state-space projection of non-oscillating circuit

Fig. 4: Oscillation results for tunnel diode oscillator

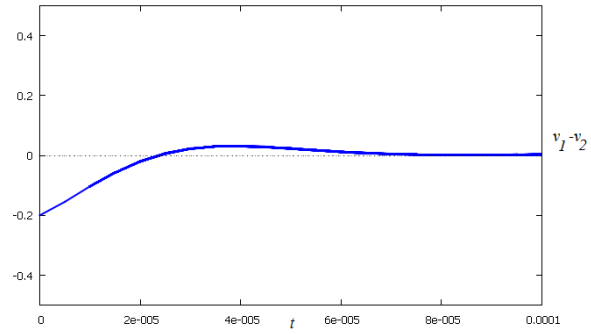
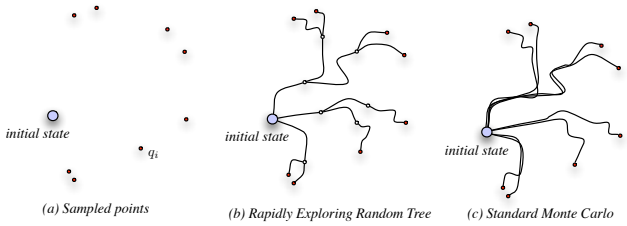


Fig. 6: The TRRT trace of signal deviation for a loop filter



(a) Sampled points (b) Rapidly Exploring Random Tree (c) Standard Monte Carlo

Fig. 7: Comparison of TRRT and Monte Carlo simulations

Carlo's linear simulation traces in a tree data structure. Figure 7 shows the TRRT versus Monte Carlo. To explore the given sampled points in the state-space, RRT shares many path, however Monte Carlo has many overlapping traces.

In the best case, if we fully populate the TRRT by running the algorithm for d^{m-1} samples, the TRRT is exponentially $(O(\frac{d^{m-1}}{m}))$ more efficient than the standard Monte Carlo algorithm. In the worst case, if we only provide m samples, each node in the final tree would have only one output, node and it would be the same as a single Monte Carlo simulation. Therefore for relatively small computational cost, TRRT has higher performance efficiency and verification utility than Monte Carlo simulations.

VI. Related Work and Conclusions

Zaki et al. survey recent literature on runtime monitoring and verification of analog and mixed-signal (AMS) designs [18]. Researchers have employed a variety of techniques to analyze the transient behaviors of circuits in either an on-line or off-line fashion. Examples of such approaches includes using interval arithmetic to validate the behavior of the circuit [17], using linear hybrid automation as a template monitor for online monitoring [4], and generating observers from PSL properties to monitor the simulation [15], [16].

The specification language we used in our work was first developed in [13], [11]. The tool described in those paper, *AMT*, synthesizes a timed automaton that monitors simulation traces for property violations. In other work, [3] propose use use repeated SPICE simulation to explore the state-space of analog circuits for all possible discrete values.

In [14], the authors propose to introduce LTL properties into RRT to verify safety properties of hybrid systems for falsification. In a similar approach, [2] and [12] use RRT to generate counter-examples in analog and hybrid systems. Recently [8], used μ-calculus to reason about RRT in discrete-time control systems.

In conclusion, we have presented a novel algorithm that uses a time-augmented rapidly-exploring random tree for incremental runtime monitoring and verification of analog circuits. Our approach is more efficient and provides more usable verification results than standard Monte Carlo simulations.

References

[1] T. Dang and T. Nahhal. Randomized simulation of hybrid systems for circuit validation. *Proceedings of Forum on Specification and Design Languages (FDL)*, 2006.

[2] T. Dang and T. Nahhal. Coverage-guided test generation for continuous and hybrid systems. *Formal Methods in System Design*, 32(2):183–213, 2009.

[3] T. R. Dastidar and P. P. Chakrabarti. A verification system for transient response of analog circuits. *ACM Transactions on Design Automation of Electronic Systems*, 12(3), 2007.

[4] G. Frehse, B. H. Krogh, and R. A. Rutenbar. Time domain verification of oscillator circuit properties. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(3):1571–0661, 2006.

[5] A. B. Grebene. The monolithic phase-locked loop: a versatile building block. *IEEE Spectrum*, 8(3):38–49, March 1971.

[6] S. Gupta, B. Krogh, and R. Rutenbar. Towards fromal verification of analog desings. *IEEE/ACM International Conference on Computer Aided Design*, pages 210–217, 2004.

[7] J. Havlicek, D. Fisman, and C. Eisner. Basic results on the semantics of Accellera PSL 1.1 foundation language. Technical report, Accelera, 2004.

[8] S. Karaman and E. Frazzoli. Sampling-based optimal motion planning with deterministic μ-calculus specifications. In *American Control Conference (ACC)*, June 2012.

[9] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, UK, 2006.

[10] S. M. Lavalle and J. J. Kuffner. Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions*, pages 293–308. A K Peters, Wellesley, MA, 2000.

[11] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. *FORMATS/FTRTFT*, 2004.

[12] T. Nahhal and T. Dang. A coverage-guided test generation tool for hybrid systems. *ARTIST WS: Tool Platforms for ES Modelling, Analysis and Validation*, 2007.

[13] D. Nickovic and O. Maler. AMT: a property-based monitoring tool for analog systems. *FORMATS*, 2007.

[14] E. Plaku, L. E. Kavraki, and M. Y. Vardi. Falsification of LTL safety properties in hybrid systems, 2009.

[15] G. A. Sammane, M. H. Zaki, Z. J. Dong, and S. Tahar. Towards assertion based verification of analog and mixed signal designs using psl. *Proceedings of Forum on Specification and Design Languages (FDL)*, 2007.

[16] Z. Wang, N. Abbasi, R. Narayanan, M. H. Zaki, G. Al Sammane, and S. Tahar. Verification of analog and mixed signal designs using online monitoring. *Proceedings of the 2009 IEEE 15th International Mixed-Signals, Sensors, and Systems Test Workshop*, 2009.

[17] M. H. Zaki, S. Tahar, and G. Bois. A practical approach for monitoring analog circuits. *Proceedings of the 16th ACM Great Lakes symposium on VLSI (GLS-VLSI'06)*, 2006.

[18] M. H. Zaki, S. Tahar, and G. Bois. Formal verification of analog and mixed signal designs: A survey. *Microelectronics Journal*, 39(12):1395–1404, 2008.