Chapter 5
Framework for Formally Verifying Analog and Mixed-Signal Designs

Mohamed H. Zaki, Osman Hasan, Sofiène Tahar and Ghiath Al-Sammane

Abstract This chapter proposes a complementary formal-based solution to the verification of analog and mixed-signal (AMS) designs. The authors use symbolic computation to model and verify AMS designs through the application of induction-based model checking. They also propose the use of higher order logic theorem proving to formally verify continuous models of analog circuits. To test and validate the proposed approaches, they developed prototype implementations in Mathematica and HOL and target analog and mixed-signal systems such as delta-sigma modulators.

Keywords Formal verification · Computer-aided analysis · Symbolic verification · Theorem proving · Electronic design automation and methodology

5.1 Introduction

Analog and mixed-signal (AMS) integrated circuits are cornerstone components used at the interface between an embedded system and its external environment [1]. As such, AMS designs are dedicated for realizing data processing functions over physical signals, such as analog to digital (A/D) and digital to analog (D/A) converters. Computer-aided design (CAD) methods have been proposed and developed

M.H. Zaki (✉) · S. Tahar · G. Al-Sammane
Concordia University, Montreal, Québec, Canada
e-mail: mzaki@ece.concordia.ca

S. Tahar
e-mail: tahar@ece.concordia.ca

G. Al-Sammane
e-mail: sammane@ece.concordia.ca

O. Hasan
National University of Sciences and Technology, Islamabad, Pakistan
e-mail: osman.hasan@seecs.nust.edu.pk

M. Fakhfakh et al. (eds.), *Computational Intelligence in Analog and Mixed-Signal (AMS) and Radio-Frequency (RF) Circuit Design*,
DOI 10.1007/978-3-319-19872-9_5

to overcome challenges in the design process of AMS designs [2, 3]. Sophisticated CAD tools and concepts are then needed to provide unique insights into the behavior and characteristics of the integrated circuits, to help the designer select best design strategies. The verification of AMS designs is one of the most important issues in their design.

In general, there exists two main approaches for validating an electronic systems with respect to a set of given properties. The first method uses monitoring with simulation to check if a property is valid or not. However, since realistic electronics systems are accepting large numbers of input combinations, it is impossible to cover all the behaviors using simulation. The major research efforts today are centered to find a cleaver way to cover most operating modes through an intelligent generation of test cases and coverage analysis. The second method, formal verification, is exploring a mathematical model of the system in order to prove the correctness of its properties. The foundations of this method are based on logic, automata, and semantics in which roots originate from computational intelligence. For digital circuits, this is applied using, for example, model checking or satisfiability-based verification. A major obstacle here is that of state explosion as the number of states of the system is exponential in the number of state variables.

However, the situation with analog and mixed-signal designs is radically different. The continuous-time behavior of analog circuits is expressed using models of differential and algebraic equations, while discrete-time behavior is described using a system of recurrence equations (SRE). In fact, closed-form solutions and systematic mathematical analysis methods for these models exist typically only for limited classes of systems. Usually, designers use differential and difference equations models more with engineering and applied mathematics tradition, not related to the careful semantics and methodological concepts developed for modeling digital concurrent systems. However, as computer systems are becoming more complex, the importance of analog components rises as AMS systems become more often integrated. The verification with simulation alone is proven not to be enough, and formal methods are advocated to occupy a complementary currently used design methods for analog systems, as they already do for digital systems.

This chapter suggests changing the strategy by tackling the problem from the point of view of difference equations (DE) used to describe the discrete-time behavior of AMS designs. In fact, a basic understanding of discrete-time behavior is essential in the design of modern AMS designs. For instance, discrete-time signal processing is used in the design and analysis of data converters used in communication and audio systems. Moreover, discrete-time processing techniques based on switched capacitor methods are used extensively in the design of analog filters [4]. We extend the definition of DE in order to represent digital components. The model then is called a generalized SRE. Then, we define the algorithms of bounded model checking (BMC) [5] on the SRE model by means of an algebraic computation theory based on interval arithmetics [6]. We associate the bounded model checking with a powerful and fully decidable equational theorem proving to verify

properties for unbound time using induction. We also propose to use higher order logic theorem proving in order to formally verify continuous models of analog circuits. In order to facilitate the user-guided verification process, we develop a library of higher order logic models for commonly used analog components, such as resistor, inductor, and capacitor, and circuit analysis laws, such as Kirchhoff's voltage and current laws. These foundations along with the formalization of calculus fundamentals can be used to reason about the correctness of any AMS property, which can be expressed in a closed mathematical form, within the sound core of a theorem prover. We illustrate the proposed method on the verification of a variety of designs including $\Delta\Sigma$ modulator and a voltage-controlled oscillator. The rest of the chapter is organized as follows: We start in Sect. 5.2 by discussing relevant related work. The bounded model checking methodology is presented in Sect. 5.3 followed by a description of the theorem proving verification methodology in Sect. 5.4 before concluding with a discussion in Sect. 5.5.

5.2 Related Work

Using formal methods, two types of properties are frequently distinguished in temporal logic: *Safety properties* state that something bad does not happen, while *liveness properties* prescribe that something good eventually happens. In the context of AMS designs, examples of safety properties can be about voltages at specific nodes not exceeding certain values throughout the operation. Such a property is important when designing AMS circuits, as a voltage exceeding a certain specified value can lead to failure of functionality and ultimately to a breakdown of the circuit which can result in undesirable consequences of the whole design. On the other hand, occurrence of oscillation or switching is good example of liveness properties. A bounded liveness property specifies that something good must happen within a given time; for example, switching must happen within n units of time, from the previous switching occurrence. This section overviews the research activities in the application of formal methods for the verification of AMS systems with respect to safety and liveness properties. A detailed literature overview of AMS formal verification can be found in [7].

Model checking and reachability analysis are proposed for validating AMS designs over a range of parameter values and a set of possible input signals. Common in these methods is the necessity for the explicit computation of the reachable sets corresponding to the continuous dynamics behavior. Such computation is usually approximated due to the difficulty to obtain exact values for the reachable state space (e.g., closed-form solutions for ODEs cannot be obtained in general). Several methods for approximating reachable sets for continuous dynamics have been proposed in the literature. They rely on the discretization of the continuous state space by using over-approximating representation domains such as polyhedra and hypercubes [8, 9]. On-the-fly algorithms have been proposed to

address shortcomings of the previous method [10–13]. The model checking tools d/dt [14], CheckMate [15], and PHaver [16] are adapted and used in the verification of a biquad low-pass filter [14], a tunnel diode oscillator and a $\Delta\Sigma$ modulator [15, 17], and voltage-controlled oscillators [16].

5.3 First Verification Methodology: Bounded Model Checking

Our methodology aims to prove that an AMS description satisfies a set of properties. This is achieved in two phases: modeling and verification, as shown in Fig. 5.1. The AMS description is composed in general of a digital part and an analog part. For the analog part, it could be described using recurrence equations. For the digital part, it could be described using event driven models. The properties are temporal relations between signals of the system. Starting with an AMS description and a set of properties, the symbolic simulator performs a set of transformations by rewriting rules in order to obtain a normal mathematical representation called generalized SRE [18]. These are combined recurrence relations that describe each property blended directly with the behavior of the system. The next step is to prove these properties using an algebraic verification engine that combines bounded model checking over interval arithmetic [6] and induction over the normal structure of the generalized recurrence equations. Interval analysis is used to simulate the set of all input conditions with a given length that drives the

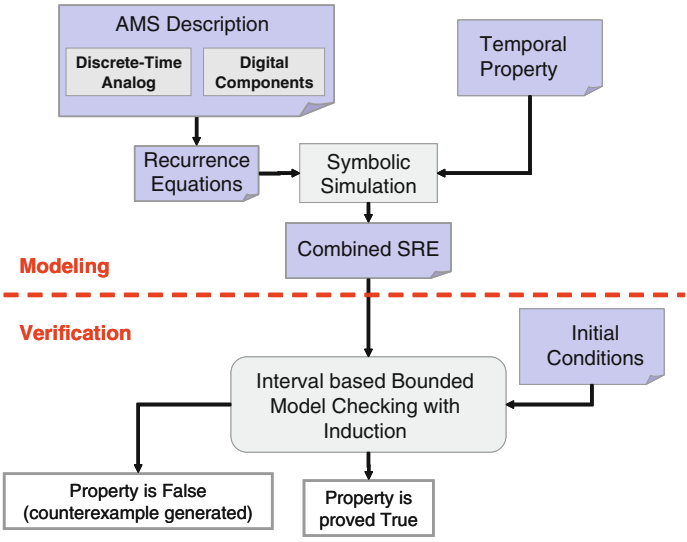

Fig. 5.1 Diagram of the bounded model checking verification

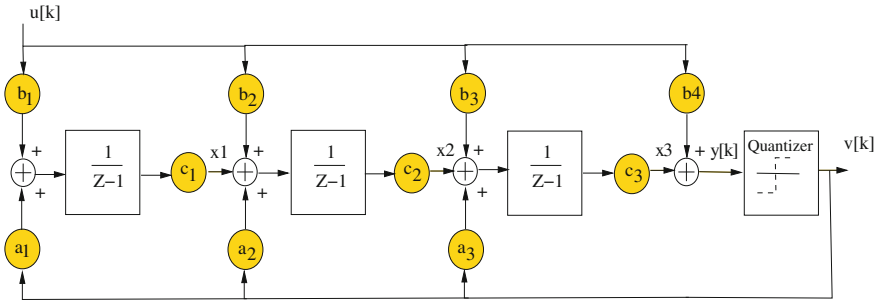

Fig. 5.2 Third-order $\Delta\Sigma$ modulator

discrete-time system from given initial states to a given set of final states satisfying the property of interest. If for all time steps the property is satisfied, then verification is ensured; otherwise, we provide counterexamples for the non-proved property. Due to the over-approximation associated with interval analysis, divergence may occur, hence preventing the desired verification. To overcome such drawback, unbounded verification can be achieved using the principle of induction over the structure of the recurrence equations. A positive proof by induction ensures that the property of interest is always satisfied; otherwise, a witness can be generated identifying a counterexample.

5.3.1 Modeling and Specification

Recurrence equations are functional models used for the definition of relations between consecutive elements of a sequence. In the current work, we argue that, for certain classes of AMS designs, it is more natural to represent their behavior using recurrence equations rather than other conventional models such as hybrid automata. The notion of recurrence equation is extended in [18] to describe digital circuits with control elements, using what is called generalized *If-formula*. Such formalization, we believe, is practical in modeling hybrid systems such as discrete-time AMS design, where discrete components control the dynamics of the circuit, for example, the valuation of an analog signal. In mathematical analysis, we define recurrence equations by:

Definition 1 (*Recurrence equation*) Let $\mathbb{K}$ be a numerical domain ($\mathbb{N}, \mathbb{Z}, \mathbb{Q}$ or $\mathbb{R}$), a recurrence equation of order $n_0 \in \mathbb{N}$ is a formula that computes the values of a sequence $U(n) \in \mathbb{K}$, $\forall n \in \mathbb{N}$, as a function of last n_0 values:

$$U(n) = f(U(n-1), U(n-2), \ldots, U(n-n_0)) \tag{5.1}$$

Definition 2 (*Generalized If-formula*) In the context of symbolic expressions, the generalized *If-formula* is a class of expressions that extend recurrence equations to describe digital systems. Let $\mathbb{K}$ be a numerical domain ($\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$ or $\mathbb{B}$), a generalized *If-formula* is one of the following:

- A variable $x_i(n) \in \mathbf{x}(n)$, with $i \in \{1, \ldots, d\}$, $n \in \mathbb{N}$, and $\mathbf{x}(n) = \{x_1(n), \ldots, x_d(n)\}$.
- A constant $C \in \mathbb{K}$
- Any arithmetic operation $\diamondsuit \in \{+, -, \div, \times\}$ between variables $x_i(n) \in \mathbb{K}$
- A logical formula: any expression constructed using a set of variables $x_i(n) \in \mathbb{B}$ and logical operators: $not, and, or, xor, nor, \ldots$, etc.
- A comparison formula: any expression constructed using a set of $x_i(n) \in \mathbb{K}$ and a comparison operator $\alpha \in \{=, \neq, <, \leq, >, \geq\}$.
- An expression $IF(X, Y, Z)$, where X is a logical formula or a comparison formula and Y, Z are any generalized *If-formula*. Here, $IF(x, y, z) : \mathbb{B} \times \mathbb{K} \times \mathbb{K} \rightarrow \mathbb{K}$ satisfies the axioms:

  1. $IF(True, X, Y) = X$
  2. $IF(False, X, Y) = Y$

Definition 3 (*Generalized SRE*) The following describes the transition relation of the system at the end of a simulation time unit n, by the way of a SRE; one equation for each element x in the system is:

$$x_i(n) = f_i(x_j(n - \gamma)), (j, \gamma) \in \varepsilon_i, \forall n \in \mathbb{Z} \tag{5.2}$$

where $f_i(x_j(n - \gamma))$ is a generalized *If-formula*. The set ε_i is a finite non-empty subset of $1, \ldots, d \times \mathbb{N}$, with $j \in \{1, \ldots, d\}$. The integer γ is called the delay.

Example 1 Consider the third-order discrete-time $\Delta\Sigma$ modulator illustrated in Fig. 5.2. Such class of $\Delta\Sigma$ design can be described using vectors recurrence equations:

$$X(k + 1) = CX(k) + Bu(k) + Av(k) \tag{5.3}$$

where A, B, and C are matrices providing the parameters of the circuit, $u(k)$ is the input signal, $v(k)$ is the digital part of the system and $b_4 = 1$. In more detail, the recurrence equations for the analog part of the system are:

$$\begin{aligned} x_1(k + 1) &= x_1(k) + b_1 u(k) + a_1 v(k) \\ x_2(k + 1) &= c_1 x_1(k) + x_2(k) + b_2 u(k) + a_2 v(k) \\ x_3(k + 1) &= c_2 x_2(k) + x_3(k) + b_3 u(k) + a_3 v(k) \end{aligned} \tag{5.4}$$

Also, the condition of the threshold of the quantizer is computed to be equal to $c_3x_3(k) + u(k)$. The digital description of the quantizer is transformed into a recurrence equation using the approach defined in [18]. Thus, the equivalent recurrence equation that describes $v(k)$ is $v(k) = IF(c_3x_3(k) + u(k) \geq 0, -a, a)$, where a is the maximum output value of the quantizer, typically equals to one.

In order to reason about the functional properties of the design under verification, we need a language that describes the temporal relations between the different signals of the system, including input, output, and internal signals. We adopt the basic subset of linear temporal logic (LTL) [19], as the specification language. Each property $P(n)$ is composed of two parts: a Boolean formula and a temporal operator. The Boolean formula $p(n)$ is a recurrence time relation written using a logical formula (see Definition 2) built over the SREs of the system. To describe properties on analog signals such as current and voltages, atomic propositions, $q(n)$, are used, which are predicates (inequalities) over reals. The provided propositions are algebraic relations between signals (variables) of the system, such that the Boolean formula is a logical combination of such atomic propositions.

Definition 4 (*Atomic Property*) An atomic property $q(n)$ is a logical formula defined as follows: $q(n) = \chi(n) \diamondsuit y$, where $\diamondsuit \in \{<, \leq, >, \geq, =, \neq\}$, $\chi(n)$ is an arithmetic formula over the design signals and y is an arbitrary value ($y \in \mathbb{R}$)

The temporal operator can be one of the basic LTL operators: Next (**X**), Eventually (**F**), and Always (**G**). As in traditional BMC, we define temporal operators regarding a bounded time step k. Thus, the verification of the temporal part is handled by the verification engine during reachability analysis.

Example 2 Consider the $\Delta\Sigma$ modulator of Example 1. The modulator is said to be stable if the integrator output remains bounded under a bounded input signal, thus avoiding the overloading of the quantizer in the modulator. This property is of a great importance since the integrator saturation can deteriorate circuit performance, hence leading to instability. If the signal level at the quantizer input exceeds the maximum output level by more than the maximum error value, a quantizer overload occurs. The quantizer in the modulator shown in Fig. 5.2 is a one-bit quantizer with two quantization levels, +1 V and −1 V. Hence, the quantizer input should be always bounded between specific values in order to avoid overloading [15]. The stability property of the $\Delta\Sigma$ modulator is written as $P(k) := \mathbf{G}p(k)$, where

$$p(k) = (x_3(k) > -2 \wedge x_3(k) < 2) \tag{5.5}$$

The symbolic simulation algorithm is based on rewriting by substitution. The computation aims to obtain the SRE defined in the previous section. In the context of functional programming and symbolic expressions, we define the following functions [20].

Definition 5 (*Substitution*) Let u and t be two distinct terms and x be a variable. We call $x \rightarrow t$ a substitution rule. We use $Replace(u, x \rightarrow t)$, read "replace in u any occurrence of x by t," to apply the rule $x \rightarrow t$ on the expression u.

The function *Replace* can be generalized to include a list of rules. *ReplaceList* takes as arguments an expression *expr* and a list of substitution rules $\{R_1, R_2, \ldots, R_n\}$. It applies each rule sequentially on the expression. $ReplaceRepeated(expr, \mathscr{R})$ applies a set of rules $\mathscr{R}$ on an expression *expr* until a fixpoint is reached, as shown in Definition 6.

Definition 6 (*Repetitive Substitution*) Repetitive substitution is defined using the following procedure:

$$
\begin{array}{l}
ReplaceRepeated(expr, \mathscr{R}) \\
\quad Begin \\
\quad\ Do \\
\qquad\qquad expr_t = ReplaceList(expr, \mathscr{R}) \\
\qquad\qquad expr = expr_t \\
\quad\ Until\ FP(expr_t, \mathscr{R}) \\
\quad End
\end{array}
$$

A substitution fixpoint $FP(expr, R)$ is obtained, if $Replace(expr, R) \equiv Replace(Replace(expr, R), R)$.

Depending on the type of expressions, we distinguish the following kinds of rewriting rules:

Polynomial Symbolic Expressions R_{Math} are rules intended for the simplification of polynomial expressions $(\mathbb{R}^n[x])$.

Logical Symbolic Expressions R_{Logic} are rules intended for the simplification of Boolean expressions and to eliminate obvious ones such as $(and(a, a) \rightarrow a)$ and $(not(not(a)) \rightarrow a)$.

If-formula Expressions R_{IF} are rules intended for the simplification of computations over *If-formulas*. The definition and properties of the *IF* rules, such as reduction and distribution, are defined as follows (see [21] for more details):

- IF Reduction: $IF(x, y, y) \rightarrow y$
- IF Distribution: $f(A_1, \ldots, IF(x, y, z), \ldots, A_n) \rightarrow IF(x, f(A_1, \ldots, y, \ldots, A_n), f(A_1, \ldots, z, \ldots, A_n))$

Equation Rules R_{Eq} result from converting other equations in the SRE into a set of substitution rules.

Interval Expressions R_{Int} are rules intended for the simplification of interval expressions.

Interval-Logical Symbolic Expressions $R_{Int-Logic}$ are rules intended for the simplification of Boolean expressions over intervals.

Rules R_{Int} and $R_{Int-Logic}$ are described in more detail later on. In the case of symbolic expressions over $\mathbb{R}$, the normal form is obtained using a Buchberger-based algorithm for the construction of the Gröbner base. The symbolic

computation uses the repetitive substitution $ReplaceRepeated(Expr, \mathscr{R})$ (defined in Definition 6) over the set of rules defined above as follows:

Definition 7 (*Symbolic Computation*) A symbolic computation over the *SREs* is defined as:

$$Symbolic\_Comp(X_i(n)) = ReplaceRepeated\big(X_i(n), R_{simp}\big)$$

where $R_{simp}(t) = R_{Math} \cup R_{Logic} \cup R_{IF} \cup R_{Eq} \cup R_{Int} \cup R_{Int-Logic}$.

The correctness of this algorithm and the proof of termination and confluence of the rewriting system formed by all above rules are discussed in [18].

Example 3 Applying Definition 6 for the $\Delta\Sigma$ modulator of Example 1, we obtain the following unified modeling for both the analog and discrete parts.

$$\begin{aligned} x_1(k+1) &= \text{if}(c_3x_3(k) + u(k) \geq 0, x_1(k) + b_1u(k) - a_1a, \\ &\qquad x_1(k) + b_1u(k) + a_1a) \\ x_2(k+1) &= \text{if}(c_3x_3(k) + u(k) \geq 0, c_1x_1(k) + x_2(k) + b_2u(k) \\ &\qquad -a_2a, c_1x_1(k) + x_2(k) + b_2u(k) + a_2a) \\ x_3(k+1) &= \text{if}(c_3x_3(k) + u \geq 0, c_2x_2(k) + x_3(k) + b_3u(k) \\ &\qquad -a_3a, c_2x_2(k) + x_3(k) + b_3u(k) + a_3a) \end{aligned} \tag{5.6}$$

The expression of the property in Example 2 after symbolic simulation is:

$$\begin{aligned} p(k+1) = \text{if}(&c_3x_3(k) + u(k) \geq 0, \\ &-2 < c_2x_2(k) + x_3(k) + b_3u(k) - a_3a, \\ &c_2x_2(k) + x_3(k) + b_3u(k) + a_3a < 2) \end{aligned}$$

5.3.2 The Automated Verification Algorithm

The proposed verification algorithm is based on combining induction and bounded model checking to generate correctness proof for the system. This method is an algebraic version of the induction-based bounded model checking developed recently for the verification of digital designs [22]. We start with an initial set of states encoded as intervals as shown in Fig. 5.3. Then, iteratively the possible reachable successors states from the previous states are evaluated using interval analysis-based computation rules over the SREs, i.e., the output of this step is a reduced *If-formula* where all variables are substituted by intervals. If there exits a path that evaluates the property to be false, then we search for a concrete counterexample. Otherwise, if all paths give true, then we transform the set of current

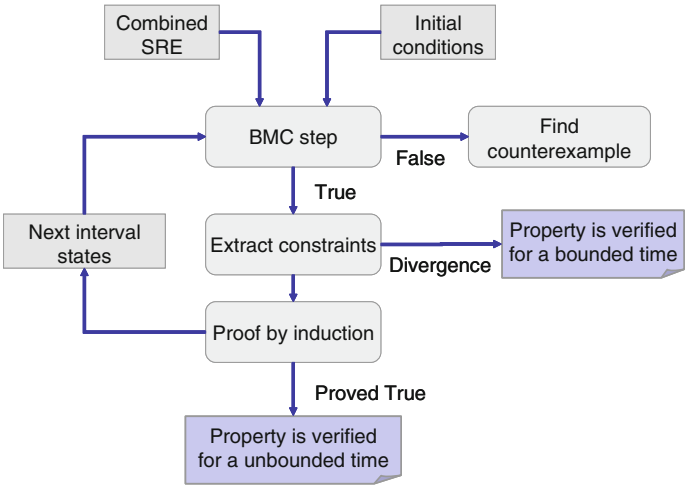

Fig. 5.3 Overview of the verification algorithm

states to constraints and we try to prove by induction that the property holds for all future states. If a proof is obtained, then the property is verified. Otherwise, if the proof fails, then the BMC step is incremented; we compute the next set of interval states and the operations are re-executed.

In summary, the verification loop terminates in one of the following situations:

- Complete Verification:
  - The property is proved by induction for all future states.
  - The property is false and a concrete counterexample is found.
- Bounded Verification:
  - The resource limits have been attained (memory or CPU) as the verification is growing exponentially with increasing number of reachability analysis steps.
  - The constraints extracted from the interval states are divergent with respect to some pre-specified criteria (e.g., width of computed interval states).

5.3.2.1 Background

Bounded Model Checking: Given a state transition system $(S, I, \mathscr{T})$, where S is the set of states, $I \subseteq S$ is the set of initial states, and $\mathscr{T} \subseteq S \times S$, the general bounded model checking problem can be encoded as follows:

$$\mathrm{BMC}(P,k) \triangleq I(s_0) \wedge \bigwedge_{i=0}^{k-1} \mathscr{T}(s_i, s_{i+1}) \rightarrow P(s_k) \tag{5.7}$$

where $I(s_0)$ is the initial valuation for the state variables, $\mathscr{T}$ defines the transition between two states, and $P(s_k)$ is the property valuation at step k. For instance,

$$P(s_k) \triangleq \mathbf{G}p(s_k) = \bigwedge_{i=0}^{k} p(s_i) \quad \text{or} \quad P(s_k) \triangleq \mathbf{F}p(s_k) = \bigvee_{i=0}^{k} p(s_i)$$

In practice, the inverse of the property ($\neg P$) under verification is used in the BMC algorithm [22], which we refer to as $\overline{BMC}$. When a satisfying valuation is returned by the solver, it is interpreted as a counterexample of length k and the property P is proved unsatisfied ($\neg P$ is satisfied). However, if the problem is determined to be unsatisfiable, the solver produces a proof (of unsatisfiability) of the fact that there are no counterexamples of length k.

Interval Arithmetics: Interval domains give the possibility to extend the notion of real numbers by introducing a sound computation framework [6]. The basic interval arithmetics are defined as follows:

Let $I_1 = [a, b]$ and $I_2 = [a', b']$ be two real intervals (bounded and closed), the basic arithmetic operations on intervals are defined by:

$$I_1 \Phi I_2 \triangleq \{r_1 \Phi r_2 | r_1 \in I_1 \wedge r_2 \in I_2\}$$

with $\Phi \in \{+, -, \times, /\}$ except that I_1/I_2 is not defined if $0 \in I_2$ [6]. In addition, other elementary functions can be included as basic interval arithmetic operators. For example, the exponential function *exp* may be defined as $exp([a, b]) = [exp(a), exp(b)]$. The guarantee that the real solutions for a given function are enclosed by the interval representation is formalized by the following property.

Definition 8 (*Inclusion Function*) [6] Let $f : \mathbb{R}^d \rightarrow \mathbb{R}$ be a continuous function, then $F : \mathbb{I}^d \rightarrow \mathbb{I}$ is an interval extension (inclusion function) of f if

$$\{f(x_1, \ldots, x_d) | x_1 \in X_1, \ldots, x_d \in X_d\} \subseteq F(X_1, \ldots, X_d) \tag{5.8}$$

where $\mathbb{I}$ is the interval domain and $X_i \in \mathbb{I}$, $i \in \{1, \ldots, d\}$.

Inclusion functions have the property to be inclusion monotonic (i.e., $X_{\mathbb{I}} \subseteq Y_{\mathbb{I}} \rightarrow F(X_{\mathbb{I}}) \subseteq F(Y_{\mathbb{I}})$), hence allowing the checking of inclusion fixpoints [6].

***d*-induction**: In formal verification, induction has been used to prove a property $\mathbf{G}P(n)$ in a transition system by showing that P holds in the initial states of the system and that P is maintained by the transition relation of the system. As such, the induction hypotheses are typically much simpler than a full reachable state description. Besides being a complete proof technique, when it succeeds, induction is able to handle larger models than bounded model checking, since the induction step has to consider only paths of length 1, whereas bounded model checking needs

to check sufficiently long paths to get a reasonable confidence. Hence, simple induction is not powerful enough to verify many properties.

d-induction [22] is a modified induction technique, where one attempts to prove that a property holds in the current state, assuming that it holds in the previous d consecutive states. Essentially, induction with depth corresponds to strengthening the induction hypothesis by imposing the original induction hypothesis on d consecutive time frames. Given a state transition system $(S, I, \mathcal{T})$, where S is the set of states and $I \subseteq S$ is the set of initial states, $\mathcal{T} \subseteq S \times S$, the d-induction proof is defined as $d\text{-Ind}_{\text{proof}} \triangleq \psi_{d-\text{base}} \wedge \psi_{d-\text{induc}}$, where $\psi_{d-\text{base}}$ is the induction base and $\psi_{d-\text{induc}}$ is the induction step defined as follows:

$$\psi_{d-\text{base}} \triangleq I(s_0) \wedge \bigwedge_{i=0}^{d-1} \mathcal{T}(s_i, s_{i+1}) \Rightarrow \bigwedge_{i=0}^{d} p(s_i)$$

and (5.9)

$$\psi_{d-\text{induc}} \triangleq \bigwedge_{i=k}^{k+d} \mathcal{T}(s_i, s_{i+1}) \wedge \bigwedge_{i=k}^{k+d} p(s_i) \Rightarrow p(s_{k+d+1})$$

It is worth noting that when $d = 1$, we have exactly the basic induction steps defined in classical induction. Similar to the general induction methods, (un) satisfiability-based induction $d\text{-Ind}_{\text{sat}}$ is the dual of the induction proof; $\text{Ind}_{\text{sat}} = \neg d\text{-Ind}_{\text{proof}}$ with $d\text{-Ind}_{\text{sat}} \triangleq \phi_{d-\text{base}} \vee \phi_{d-\text{induc}}$, where the formulas $\phi_{d-\text{base}}$ (the base step) and $\phi_{d-\text{induc}}$ (the induction step) are defined as follows:

$$\phi_{d-\text{base}} \triangleq I(s_0) \wedge \bigwedge_{i=0}^{d-1} \mathcal{T}(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{d} \neg p(s_i)$$

and (5.10)

$$\phi_{d-\text{induc}} \triangleq \bigwedge_{i=k}^{k+d} \mathcal{T}(s_i, s_{i+1}) \wedge \bigwedge_{i=k}^{k+d} p(s_i) \wedge \neg p(s_{k+d+1})$$

The advantage of d-induction over classical induction is that it provides the user with ways of strengthening the induction hypothesis by lengthening the time steps d computed. Practically speaking, $\phi_{d-\text{base}}$ is bounded model checking ($\overline{BMC}$) as defined earlier in this section. For the case of systems with variables interpreted over real domains such as AMS designs, the satisfiability of the formulas with a given set of initial conditions requires algorithms to produce bounded envelopes for all reachable states at the discrete-time points. In the following, we demonstrate how to achieve BMC using interval arithmetics.

5.3.2.2 BMC Realization

The bounded forward reachability algorithm starts at the initial states and at each step computes the image, which is the set of reachable interval states. This procedure is continued until either the property is falsified in some state or no new

states are encountered. We evaluate the reachable states over interval domains, at arbitrary time steps. The verification steps for safety properties are shown in Algorithm 5.1. The AMS model, described as a set of recurrence equations, is provided along with the (negated) property $\neg P(n)$ under verification. Initial and environment constraints *Env_Const* are also defined prior to the verification procedure described in lines (1–12) as a loop for $N_{\max}$ time steps. At each step n, we check whether the property is satisfied or not (line 2). If $\neg P(n)$ is satisfied, then a counterexample is generated (line 9); if not, then we check if fixpoint inclusion is reached (line 3); otherwise, we update the reachable states (line 11) and go to the next time step of verification. The functions *Prop_Check*, *Find_Counterexample*, and *Update_Reach* are described below.

Algorithm 5.1 Safety BMC

Require: $x[n]$
Require: $\neg P(x[n])$
Require: $\mathscr{R}^0 = S_0$
Require: *Env_Const*
1: **for** $n = 1$ to N_{max} **do**
2: **if** $Prop\_Check(\neg P[n], x[n]) == False$ **then**
3: **if** $Reach[T_{o_t,x[n]}] \subseteq \mathscr{R}^{n-1}$ **then**
4: **return** fixpoint reached
5: **else**
6: $Inc\_Step(n)$
7: $\mathscr{R}^{n-1} = Update\_Reach(\mathscr{R}^{n-2}, Reach[x[n-1]])$
8: **end if**
9: **else**
10: $Find\_Counterexample(\neg P[n], x[n], Env\_Const)$
11: **end if**
12: **end for**

Prop_Check: Given the property $\neg P$, apply algebraic decision procedures to check for satisfiability. The safety verification at a given step n can be defined with the following formula:

$$Prop\_Check \triangleq \mathbf{x}[n] = f(\mathbf{x}[n-1]) \wedge \neg P(\mathbf{x}[n]) \wedge x[n-1] \in \mathbb{I}^d \tag{5.11}$$

Update_Reach(R_1, R_2): This function returns the union of the states in the sets R_1 and R_2.

Reach[$x[n]$:] This evaluates the reachable states over interval domains, at an arbitrary time step.

Find_Counterexample($\neg P(n), x[n], Env\_Const$): This function returns a counterexample, indicating a violation of the property, within the environment constraints.

Setting bounds on the maximum number of iterations ensures that the algorithm will eventually terminate in one of the following possibilities. If at a given time step $n \leq N_{\max}$, no new interval states are explored, then fixpoint inclusion guarantees

that the property will be always verified; otherwise, if a property is proved to be incorrect, then a counterexample is generated. If we reach the maximum number of steps $n = N_{\max}$, and no counterexample is generated, then the property is verified up to bounded step $N_{\max}$.

Example 4 Given the design in Example 1 and the safety property in Example 2, we apply Algorithm 5.1. For instance, the correctness of the property $P(k+1)$ (see Example 3) depends on the parameter vectors A, B, and C, the values of variables $x_1(k)$, $x_2(k)$, and $x_3(k)$, the time k, and the input signal $u(k)$ (see Table 5.1). We verify the $\Delta\Sigma$ modulator for the following set of parameters inspired from the analysis in [15]:

$$\begin{cases} a = 1 & a_1 = 0.044 & a_2 = 0.2881 \\ a_3 = 0.7997 & b_1 = 0.07333 & b_2 = 0.2881 \\ b_3 = 0.7997 & c_1 = c_2 = c_3 = 1 \end{cases}$$

The initial constraints define the set of test cases over which interval-based simulation is applied. If the property is *false*, as in the first and third cases in Table 5.1, then the verification is completed and a counterexample is generated from the simulated intervals. On the contrary, when the property is *true*, we have a partial verification result as it is bounded in terms of simulation steps. The second case in Table 5.1 illustrates such limitation.

Unfortunately, we note that in some cases (as case 4 in Table 5.1), divergence happens quickly, so we cannot deduce useful information on the property. We

Table 5.1 Verification results for $\Delta\Sigma$ modulator in Example 4

Initial constraints	Property evaluation for $n = 0$ to $N_{\max}$ cycles	Counterexample	CPU time used (s)
$0.028 \le x_1(0) \le 0.03$ $-0.03 \le x_2(0) \le -0.02$ $0.8 \le x_3(0) \le 0.82, u := 0.8$	$N_{\max} = 40$ $n = 0$ to 15 true $n > 15$ false	$x_1[16] \mapsto 0.263$ $x_2[16] \mapsto 1.256, x_3[16] \mapsto 2.42$	1.5
$0.012 \le x_1(0) \le 0.013$ $0.01 \le x_2(0) \le 0.02$ $0.8 \le x_3(0) \le 0.82, u := 0.54$	$N_{\max} = 38$ true		31
$0.163 \le x_1(0) \le 0.164$ $-0.022 \le x_2(0) \le -0.021$ $0.8 \le x_3(0) \le 0.82, u := 0.6$	$N_{\max} = 40$ $n = 0$ to 17 true $n > 17$ false	$x_1[19] \mapsto 0.163$ $x_2[19] \mapsto 0.886, x_3[19] \mapsto 2.47$	0.8
$0.012 \le x_1(0) \le 0.013$ $0.01 \le x_2(0) \le 0.02$ $0.8 \le x_3(0) \le 0.82, 0.58 \le u \le 0.6$	Divergent at time step 4		0.5

tackle such problem by extending the bounded model checking with an induction engine as proposed in the verification methodology.

5.3.2.3 Constrained Induction-Based Verification

In the following, we define an induction engine over the SREs for the safety property verification of AMS designs. The inductive proof, which is a special case of the *d*-induction described earlier in this chapter, for verifying a safety property $P(n) = \mathbf{G}p(n)$, can be derived by checking the formula $\text{Ind}_{\text{proof}} \triangleq \psi_{\text{base}} \wedge \psi_{\text{induc}}$, where ψ_{base} is the induction base and ψ_{induc} is the induction step defined as follows:

$$\psi_{\text{base}} \triangleq \forall s \in S_0 : I(s_0) \Rightarrow p(s_0)$$
and
$$\psi_{\text{induc}} \triangleq \forall s_k, s_{k+1} \in S : \mathscr{T}(s_k, s_{k+1}) \wedge p(s_k) \Rightarrow p(s_{k+1}) \tag{5.12}$$

The core of the induction engine is a decision procedure function that checks satisfiability of algebraic formulas under certain constraints on quantified state variables.

Definition 9 (*The Prove Function*)

$$\begin{aligned} &Prove(quant(X, cond, expr)) = \\ &\quad If(Prop\_Verify(quant(X, cond, expr))) = True, \\ &\quad\quad True, \\ &\quad\quad Find\_Counterexample(cond \wedge \neg expr) \end{aligned}$$

The decision procedure function *Prove* tries to prove a property of the form $quant(X, cond, expr)$, using *Prop_Verify*; otherwise, it gives a counterexample using *Find_Counterexample*, where $quant \in \{\forall, \exists\}$ define quantifiers over a set of state variables x, *cond* is a logical combination of comparison formulas constructed over the variables x describing initial and environment constraints, and *expr* is an *If-formula* expression representing the property of interest, obtained after applying the symbolic rule outlined earlier. Similar to *Prop_Check*, *Prop_Verify* applies algebraic decision procedures to check for satisfiability, but for all time steps n. The safety verification can be defined with the following formula:

$$Prop\_Verify \triangleq \forall n \cdot (\mathbf{x}[n] = SRE(x[n])) \wedge P(\mathbf{x}[n]) \tag{5.13}$$

The *Prove* function generates a counterexample if the property of interest cannot be proved to hold using $Find\_Counterexample(cond \wedge \neg expr)$. If a proof cannot be obtained, then we may need to find a particular combination of inputs and local signal values for which the property is not satisfied. The properties verification

using *Prove* starts by checking the validity at time $t = 1$ and then at time $t = n$ assuming that the properties are satisfied at time $t = n - 1$. Case splitting divides the property into subproperties for which validation results are conjuncted to check the validation of the original property.

Let P be a property of the form $quant(X, cond, expr)$. We define the function *SplitProve* that depending on the *If-formula* structure of *expr*, applies the function *Prove*, or splits the verification. *SplitProve* is defined recursively as follows:

Definition 10 (*The SplitProve Function*) According to the nature of *expr*, *SplitProve* can be one of the following:

- *expr* is a comparison formula C, $SplitProve(quant(X, cond, C)) = Prove(quant(X, cond, C))$
- *expr* is a logical formula of the form $a \diamondsuit b$, with $\diamondsuit \in \{\neg, \wedge, \vee, \oplus, \ldots\}$ and a, b are *If-formulas* that take values in $\mathbb{B}$.
  $SplitProve(P)) \simeq SplitProve(quant(X, cond, a)) \diamondsuit SplitProve(quant(X, cond, b))$
- *expr* is an expression of the form $IF(q, l, r)$ $SplitProve(P) = SplitProve(quant(X, cond \wedge q, l)) \vee SplitProve(quant(X, cond \wedge \neg q, r))$

Let $P(n)$ be the recurrence equation of the property P written as an *If-formula*, $cond_{n_0}$ the initial condition at time n_0, $cond_n$ the constraints that are true for all $n > n_0$, and X the set of dependency variables of $P(n)$, and the proof by induction over n is defined as follows:

Definition 11 (*Proof by Induction*)

$$SplitProve(ForAll(X_{n_0}, cond_{n_0}, P(n_0)))$$
$$\wedge$$
$$SplitProve(ForAll(n > n_0 \wedge X_n, n \in \mathbb{N} \wedge cond_n \wedge P(n), P(n+1)))$$

Example 5 We verify the $\Delta\Sigma$ modulator of Example 1 for two sets of parameters inspired from the analysis in [15]:

$$\text{Param}_1\text{:} \begin{cases} a = 1 & a_1 = 0.044 & a_2 = 0.2881 \\ a_3 = 0.7997 & b_1 = 0.044 & b_2 = 0.2881 \\ b_3 = 0.7997 & c_1 = c_2 = c_3 = 1 & \end{cases}$$

$$\text{Param}_2\text{:} \begin{cases} a = 1 & a_1 = 0.044 & a_2 = 0.2881 \\ a_3 = 0.7997 & b_1 = 0.07333 & b_2 = 0.2881 \\ b_3 = 0.7997 & c_1 = c_2 = c_3 = 1 & \end{cases}$$

We apply induction in order to verify the $\Delta\Sigma$ modulator stability for the above sets of parameters and for two cases of conditions (state space constraints). Table 5.2 summarizes the verification results. The property is *True* if it is proved

Table 5.2 Verification results for ΔΣ modulator in Example 5

State space constraints	Property with Parameter$_1$	Property with Parameter$_2$	
Case 1	Values at $t = 0$	True	True
	$0 \le x_1(0) \le 0.01$ $-0.01 \le x_2(0) \le 0$ $0.8 \le x_3(0) \le 0.82,\ u := 0.6$		
	Values at $t = n$		
	$-0.1 \le x_1(n) \le 0.1$ $-0.5 \le x_2(n) \le 0.5$ $0.5 \le x_3(n) \le 1.5, u := 0.6$		
Case 2	Values at $t = 0$	False	False
	$0 \le x_1(0) \le 0.02$ $-0.03 \le x_2(0) \le -0.01$ $1 \le x_3(0) \le 1.4, u := 0.8$		
	Values at $t = n$	$x_2[k] \mapsto 0.4237$ $x_3[k] \mapsto 1.8378$	$x_2[k] \mapsto 0.2103$ $x_3[k] \mapsto 2$
	$-0.1 \le x_1(n) \le 0.1$ $-1 \le x_2(n) \le 0.5$ $-1 \le x_3(n) \le 2.5, u := 0.8$		

under the set of conditions and the set of parameters for all $k > 0$. If there is no k for which the property is valid, then it is *False*, and a counterexample is provided. When the property is valid for some values of k and not for other values, we say that the property is not proved and counterexamples are provided for both cases.

5.3.2.4 Combining *d*-Induction and Interval-Based BMC

The d-induction-based verification algorithm is an incremental algorithm, where depth is incremented at each step and induction is applied on the new formulas until a d-length counterexample is generated or the property is proved. The verification steps are given in Algorithm 5.2.

The AMS model, described as a set of recurrence equations, is provided along with the (negated) property $\neg P(n)$ under verification. Initial and environment constraints are also defined prior to the verification procedure described in lines (1–18) as a loop of depth $N_{\max}$ steps. For each depth $d < N_{\max}$, we first check the initial d-induction step by verifying whether the property is verified for all steps up to this depth d (line 3). If the property is false, we generate a counterexample (lines 4). Before checking the induction step (line 10), we verify whether an inclusion fix-point is reached. If so, the verification ends as it will be trivial to check for the induction step as no new verification information can be implied. When we apply the induction step, where either the property is verified for unbounded time (line 11), otherwise, we conclude that the current depth is not enough to verify the property and the depth is incremented (line 14).

Algorithm 5.2 d-induction based BMC

Require: $x[n] := SRE(\mathscr{A})$
Require: $\neg P(x[n])$
Require: $\mathscr{R}^0 = S_0$
Require: Env_Const
1: initialize $d = 1$
2: **for** $d = 1$ to N_{max} **do**
3: **if** $Prop\_Check(\neg \bigwedge_{i=0}^{d} P(i), x[n]) == True$ **then**
4: $Find\_Counterexample(\neg P(n), x[n], Env\_Const)$
5: **else**
6: **if** $Prop\_Check(\neg P(d), x[d]) == False$ **then**
7: **if** $Reach[x[d]] \subseteq \mathscr{R}^{d-1}$ **then**
8: **return** fixpoint reached
9: **else**
10: **if** $Prop\_Verify(\neg \bigwedge_{i=n}^{d+n} P(i), \bigwedge_{i=n}^{d+n} x[i]) == False$ **then**
11: return verified
12: **end if**
13: **end if**
14: $Inc\_Step(d)$
15: $\mathscr{R}^{n-1} = Update\_Reach(\mathscr{R}^{n-2}, Reach[x[n-1]])$
16: **end if**
17: **end if**
18: **end for**

It is worth noting that constraints used in the induction steps are extracted from the previous reachable states. Hence, we strengthen the induction hypothesis by lengthening the time steps d computed. In case a counterexample needs to be generated, the extracted constraints allow for finding a partial path violating the property. Setting bounds on the maximum number of iterations ensures that Algorithm 5.2 will eventually terminate in one of the following possibilities. If the initial induction step fails, a counterexample is generated; otherwise, if at a given time step $n \leq N_{\max}$, no new interval states are explored, and then, fixpoint inclusion guarantees that the property will be always verified. In case the induction step is verified true, then the algorithm terminates; otherwise, we increase the induction depth and restart the verification. If we reach the maximum number of steps $n = N_{\max}$, and no counterexample is generated, then the property is verified up to bounded step $N_{\max}$.

5.3.3 Applications

We have implemented a prototype for the presented verification algorithms using symbolic algebraic manipulation and real number theorem proving developed inside the computer algebra tool *Mathematica* [23].

5.3.3.1 Third-Order ΔΣ Modulator

We extended the verification results outlined throughout the chapter and summarized in Tables 5.1 and 5.2 by applying the *d*-induction algorithm to verify the stability of the third-order ΔΣ modulator for different combinations of design parameters, inputs, and initial conditions. We are able to prove properties using the inductive BMC method, which we were unable to verify previously using the conventional BMC method (rows 2 and 4 in Table 5.1). In row 2 (Table 5.1), we are able only to verify the property for a bounded time step, with the *d*-induction BMC method; however, we are able to prove that the property will always hold (second row with param_2 in Table 5.3). On the other hand, in row 4 (Table 5.1), the divergence occurs quickly; however, the property is proven *True* as shown in Table 5.3, row 4 with param_2.

5.3.3.2 Voltage-Controlled Oscillator

Recurrence equations have been proposed as a simplified operational modeling framework for certain AMS designs, in which precise continuous-time modeling poses challenging requirements to achieve simulation. As an instance, precise PLL verification necessitates the accounting for different time constants which render the simulation hard to achieve. Accordingly, at the early steps of the design, a discrete-time model is constructed representing the main functional aspects of the design. This can be later translated to a more refined model at subsequent design stages.

Table 5.3 *d*-induction BMC verification results for the third-order ΔΣ modulator

	State space constraints	Verification results	Verification details
Param_1	$0 \le x_1(0) \le 0.01$ $-0.01 \le x_2(0) \le 0$ $0.8 \le x_3(0) \le 0.82, u := 0.6$	Proved true by *d*-induction	*k*-step = 3
	$0 \le x_1(0) \le 0.02$ $-0.03 \le x_2(0) \le -0.01$ $1 \le x_3(0) \le 1.4, u := 0.8$	Proved true by BMC then divergent	*k*-step = 14
Param_2	$0 \le x_1(0) \le 0.01$ $-0.01 \le x_2(0) \le 0$ $0.8 \le x_3(0) \le 0.82, u := 0.6$	Proved true by *d*-induction	*k*-step = 3
	$0.012 \le x_1(0) \le 0.013$ $0.01 \le x_2(0) \le 0.02$ $0.8 \le x_3(0) \le 0.82, u := 0.54$	Proved true by *d*-induction	*k*-step = 3
	$0 \le x_1(0) \le 0.02$ $-0.03 \le x_2(0) \le -0.01$ $1 \le x_3(0) \le 1.4, u := 0.8$	Proved false by counterexample	*k*-step = 16
	$0.012 \le x_1(0) \le 0.013$ $0.01 \le x_2(0) \le 0.02$ $0.8 \le x_3(0) \le 0.82, 0.58 \le u \le 0.6$	Proved true by *d*-induction	*k*-step = 3

In the following, we apply the induction-based verification for the voltage-controlled oscillator (VCO) block of a charge pump PLL. A VCO is an oscillator, in which output frequency is controlled and varied by the applied input voltage. The recurrence equation modeling of the VCO is based on the circuit shown in Fig. 5.4 that describes a relaxation oscillator, in which output is a digital signal [4]. In the shown design, the input voltage is used to derive the VCO which according to some switching conditions triggers the one-shot timer, which in turn acts by controlling the discharging switch S_{osc} and the input to the toggle circuit. For instance, assume that the capacitor C_{osc} is initially discharged, it will be slowly charged by the current I_{osc} with the voltage V_2 at each analysis step. Once the voltage V_{th2} across the capacitor C_{osc} exceeds the threshold voltage V_{th2}, then the output of the comparator goes to high (if it is not) and the one-shot timer is activated. The details about the functionality modeling of this VCO can be found in [4].

For correct operation of the VCO within the PLL design, it is required that the output will toggle from time to time (frequency of toggling is depending on the input voltage to the VCO). Such property has a flavor of liveness characteristics, which cannot be checked directly through induction. However, we use induction to check whether the input voltage variations will not lead to improper functionality. The verified property can be stated as follows: For a given set of input voltage variations, V_{osc} will always remain unchanged ($\mathbf{G}V_{\text{osc}}[n] - V_{\text{osc}}[n-1] = 0$). If this property is verified true, then we deduce that our choice of input signal range and/or parameters values is inappropriate for a correct behavior for the design.

We verified the property over several input signal V_1 ranges, for different values of the transcendence of the VCO G_{osc} and the capacitor C_{osc}. The results in the experiments are obtained using the parameters proposed in [4]. First, we choose the range of input voltage as the interval [0, 2] volts. The property in this case is verified true. However, when we increase the input range to [0, 2.3], the property becomes false. From those two results, we deduce that a possible correct functionality would require at least a larger swing for the input signal to the VCO. In another experiment, we preserve the first input voltage range while perturbing the set of parameter values and the property is verified again to false. Another interesting property we checked is the following safety criteria: For all possible input

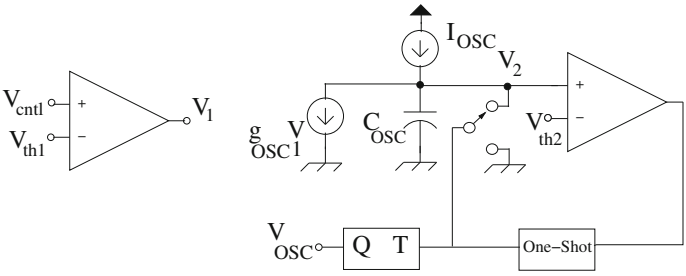

Fig. 5.4 Voltage-controlled oscillator

voltage ranges (i.e., $V_1 \in [-2.5, 2.5]$), the comparator input voltage V_2 will never exceed certain bounds (i.e., $V_2 \in [-2.51, 2.51]$). This property is verified true. In fact, this verification is very beneficial as it provides us with the knowledge of the upper and lower bounds of the reachable state space. It is important to note that the correct functionality of the VCO requires the analysis over different voltage changes and notice the output. This would demand a dynamic verification method such as reachability or simulation, rather than a static method such as induction-based verification. Nevertheless, this latter technique allows the designer to have a better knowledge about the design limitations and to avoid and prune out undesirable constraints and parameters values when integrating the design with other components.

5.4 Second Verification Methodology: Theorem Proving

Most of the existing formal verification approaches work with abstracted discretized models of analog circuits (e.g., [24, 25]). This is mainly because of the inability to model and analyze continuous systems by the widely used formal verification techniques, such as model checking or automated theorem proving. Thus, despite the inherent soundness of formal verification methods, such analysis cannot be termed as absolutely accurate. Higher order logic theorem proving can be used to overcome these limitations due to the high expressiveness of the underlying logic. However, most of the existing higher order logic theorem proving-based analog circuit verification works (e.g., [26–28]) use discrete models of the given analog circuits by abstracting the continuous details. Thus, neither real numbers nor differential equations are used to represent the analog circuit behaviors in these analyses, which makes them prone to round-off and approximation errors.

We argue that the high expressibility of higher order logic can be leveraged upon to formalize the continuous models of analog circuit implementations and their desired specifications. Their equivalence can then be verified within the sound core of a theorem prover. Due to the high expressibility of higher order logic, the proposed approach is very flexible in terms of analyzing a variety of analog circuits and reasoning about their generic properties.

There are two main challenges in the proposed approach. Firstly, due to the undecidable nature of higher order logic, the proofs have to be done interactively, which may become very tedious due to the involvement of continuous elements and transcendental functions. Secondly, no closed-form solutions exist for a large number of analog circuits, and thus for these kinds of circuits, we cannot formally reason about approximate solutions in a theorem prover. We overcome both of these challenges in the proposed methodology [29], depicted in Fig. 5.5, by developing a library of analog circuit analysis definitions, theorems, and automatic simplifiers to minimize the user effort in the formal reasoning process and by using the support of computer algebra system for solving differential equations for which no closed-form solutions exist.

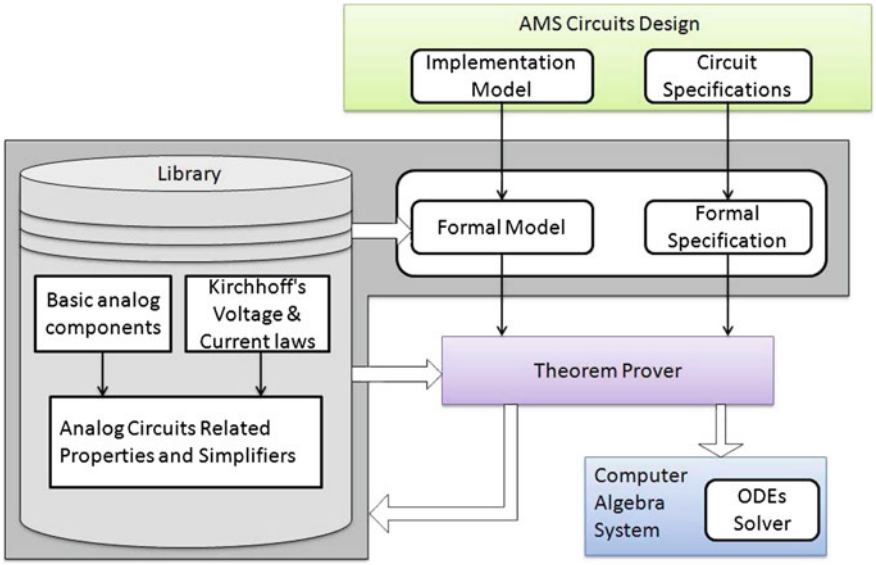

Fig. 5.5 Proposed methodology for the formal verification of AMS circuits

The first step in the proposed methodology is to obtain an implementation model of the given analog circuit by using the behavior of its individual components and its overall structure. To facilitate this formalization, we developed a database of formal definitions of commonly used analog components, such as resisters, capacitors, and inductors, and circuit analysis laws such as Kirchhoff's voltage and current laws. The second step in the proposed methodology is to develop a formal model of the specification of the circuit, which is usually expressed as a differential equation. For this purpose, we choose the HOL4 theorem prover to implement the proposed methodology since it provides formalized libraries of real numbers and calculus foundations [30]. The third step is to verify the equivalence or implication relationships between the formalized implementations and specifications. To minimize the user interaction, required in this step, we formally verified most of the frequently used properties and developed some simplifying tactics with access to these results so that the users can verify most of the proof goals associated with analog circuit verification with minimal interaction. The main contribution in this regard is the formal verification of properties related to solutions of differential equation. Finally, if the differential equation corresponding to the given analog circuit does not have a closed mathematical solution, then it can be fed to a computer algebra system, such as Mathematica, to obtain its approximate solution. It is important to note here that the soundness of the analysis is not compromised at all by the computer algebra system link since it would only be invoked for the cases where a closed-form precise solution cannot be attained.

The main strengths of the proposed approach include its generic nature and accuracy. Any kind of analog circuit can be modeled, and its corresponding linear, nonlinear, homogenous, or non-homogenous differential equation can be formally expressed in higher order logic. If a closed-form solution for this equation exists, then it can be formally verified within the sound core of a theorem prover. In this case, modeling or analysis does not involve computer arithmetics or any discretization and thus, actual continuous models are formally verified. On the other hand, if a closed form does not exist, then the analysis is done using computer algebra systems, which is definitely the most accurate method in this scenario.

In the rest of this section, we first provide a formalization of the solutions of the second-order homogeneous linear differential equation to be able to reason about the solutions of differential equations for which a closed-form solution exist. Many interesting analog circuits lead to these kinds of equations. The formalization of circuit analysis fundamentals, i.e., KVL, KCL, and basic circuit components, is provided next. A couple of illustrative examples are then presented in the end.

5.4.1 Second-Order Homogeneous Linear Differential Equations

Second-order homogeneous linear differential equations are widely used to model analog circuits, and differential equations of higher order are seldom required in this domain. They can be mathematically expressed as follows:

$$p_2(x)\frac{\mathrm{d}^2y(x)}{\mathrm{d}x} + p_1(x)\frac{\mathrm{d}y(x)}{\mathrm{d}x} + p_0(x)y(x) = 0 \tag{5.14}$$

where terms p_i represent the coefficients of the differential equation defined over a function y. The equation is linear because (i) the function y and its derivatives appear only in their first power and (ii) the products of y with its derivatives are also not present in the equation. By finding the solution of the above equation, we mean to find functions that can be used to replace the function y in Eq. (5.14) and satisfy it.

We proceed to formally represent Eq. (5.14) by first formalizing an nth-order derivative function as follows [31]:

Definition 12 (*Nth-order Derivative of a Function*)

```
⊢ (∀ f x. n_order_deriv 0 f x = f x) ∧
  (∀ f x n. n_order_deriv (n+1) f x =
      n_order_deriv n (deriv f x) x)
```

The function `n_order_deriv` accepts an integer n that represents the order of the derivative, the function f that represents the function that needs to be

differentiated, and the variable x that is the variable with respect to which we want to differentiate the function f. The function `deriv` accepts two parameters f and x and returns the derivative of the function f at point x. Thus, the function `n_order_deriv` returns the nth-order derivative of f with respect to x. Now, based on this definition, we can formalize the left-hand side (LHS) of an nth-order differential equation in HOL4 as the following definition [31].

Definition 13 (*LHS of a Nth-order Differential Equation*)

```
⊢ ∀ P y x. diff_eq_lhs P y x =
    sum(0,LENGTH P)(λn.(EL n P x) *
         (n_order_deriv n y x))
```

The function `diff_eq_lhs` accepts a list P of coefficient functions corresponding to the p_i's of Eq. (5.14), the differentiable function y and the differentiation variable x. It utilizes the functions `sum (0,m) f` and `EL m L`, which correspond to the summation ($\sum_{i=0}^{m-1} f_i$) and the mth element of a list L_m, respectively. It generates the LHS of a differential equation of order equal to the number of elements in the coefficient list P using the length of the list function `LENGTH`.

If the coefficients p_i's of Eq. (5.14) are constants, then using the fact that the derivative of the exponential function $y = e^{rx}$ (with a constant r) is a constant multiple of itself $dy/dx = re^{rx}$, we can obtain the following solution of Eq. (5.14):

$$Y(x) = c_1 e^{r_1 x} + c_2 e^{r_2 x} \tag{5.15}$$

where c_1 and c_2 are arbitrary constants and r_1 and r_2 are the roots of the auxiliary equation $p_2 r^2 + p_1 r^1 + p_0 = 0$. In this chapter, we formally verify this result which plays a key role in formal reasoning about the solutions of second-order homogeneous linear differential equations [31].

Theorem 1 *Differential Equation with distinct roots*

```
⊢ ∀ a b c c1 c2 r1 r2 x.
   (c + (b * r1) + (a * r1²) = 0) ∧
   (c + (b * r2) + (a * r2²) = 0) ⇒
   (diff_eq_lhs (const_list [c; b; a])
     (λx. c1 * (exp (r1 * x)) +
          c2 * (exp (r2 * x))) x = 0)
```

where $[c; b; a]$ represents the list of constants corresponding to the coefficients p_0, p_1, and p_2 of Eq. (5.14); r_1 and r_2 represent the roots of the corresponding auxiliary equation as given in the assumptions; c_1 and c_2 are the arbitrary constants; and x is the variable of differentiation. The function `const_fn_list` transforms a list of

real numbers to the corresponding list of constant functions recursively, i.e., functions with data type `real` → `real` that return a constant value for all values of arguments [31]. The formal reasoning about Theorem 1 is primarily based on the linearity property of higher order derivatives

5.4.2 Kirchhoff's Voltage and Current Laws

Kirchhoff's voltage law (KVL) and Kirchhoff's current law (KCL) form the most foundational circuit analysis laws. The KVL and KCL state that the directed sum of all the voltage drops around any closed network (loop) of an electrical circuit and the directed sum of all the branch currents leaving an electrical node is zero, respectively. Mathematically,

$$\sum_{k=1}^{n} V_k = 0, \ \sum_{k=1}^{n} I_k = 0 \tag{5.16}$$

where V_k and I_k represent the voltage drops across the kth component in a loop and the current leaving the kth branch in a node, respectively. The formalization is as follows [29]:

Definition 14 (*Kirchhoff's Voltage and Current Law*)

```
⊢ ∀ V t. kvl V t =
   (∀ x. 0 < x ∧ x < t ⇒
     (sum (0,LENGTH V) (λn. EL n V x) = 0))
⊢ ∀ V t. kcl I t =
   (∀ x. 0 < x ∧ x < t ⇒
     (sum (0,LENGTH I) (λn. EL n I x) = 0))
```

The function `kvl` accepts a list V of functions of type (`real` → `real`), which represents the behavior of time-dependant voltages in the given circuit and a time variable t as a *real* number. It return the predicate that guarantees that the sum of all the voltages in the loop is zero for all time instants in the interval $(0, t)$. Similarly, the function `kcl` accepts a list I, which represents the behavior of time-dependant currents and a time variable t and returns the predicate that guarantees that the sum of all the currents leaving the node is zero for all time instants in the interval $(0, t)$.

We now present some of the foundational formalization that is required to formally model analog circuits. The *V–I* characteristics of fundamental analog components such as resistors, inductors, capacitors, and op-amps can be formalized as [29]:

Definition 15 (*Resistor, Inductor Capacitor, and Op-amp*)

```
⊢ ∀ R i.resistor_voltage R i = (λt.i t * R)
⊢ ∀ R v.resistor_current R v = (λt.v t / R)
⊢ ∀ L i.inductor_voltage L i =
    (λt. L * deriv i t)
⊢ ∀ L v Io.inductor_current =
    (λt. Io + 1/L * integral (0,t) v)
⊢ ∀ C i Vo. capacitor_voltage C i Vo =
    (λt. Vo + 1/C * integral (0,t) i)
⊢ ∀ C v. capacitor_current =
    (λt. C * deriv v t)
⊢ ∀ Vpos Vneg A. op_amp_voltage Vpos Vneg A=
    (λt. A * (Vpos t - Vneg t))
```

The variables *i* and *v* represent the time-dependant current and voltage variables, respectively, in the above function definitions. While the variables *R*, *L*, and *C* represent the constant resistance, inductance, and the capacitance of their respective components, respectively. The variables *Io* and *Vo* are used in the definitions of inductance and capacitance to model the initial current in the inductor and the initial voltage across the capacitor, respectively. The parameters `Vpos, Vneg,` and `A` represent non-inverting input, inverting input, and gain of an op-amp, respectively. The function `deriv` accepts two parameters *f* and *x* and returns the derivative of the function *f* at point *x*. Likewise, the function `integral` takes three parameters *f*, *a*, and *b* and returns the integrated result of *f* in the interval (*a*, *b*). All these functions return a (`real` → `real`) type function that models the corresponding time-dependant voltage or current.

5.4.3 Applications

5.4.3.1 RLC Series Circuit

Serially connected resistor (R), inductor (L), and capacitor (C), or the RLC, circuit is one of the classical examples of an AMS circuit. It is also widely used in modeling parasitics in the metal interconnect of submicrometer ICs. We utilize the foundational formalization for analyzing AMS circuits, described in the last two subsections, to formally verify the electrical current flow relationship in the RLC circuit, shown in Fig. 5.6, with the intent to demonstrate the proposed methodology for formally analyzing AMS circuits.

The first step in the proposed methodology is to model the behavior of the given circuit in higher order logic. The behavior of the given circuit can be captured using the KVL as follows [31]:

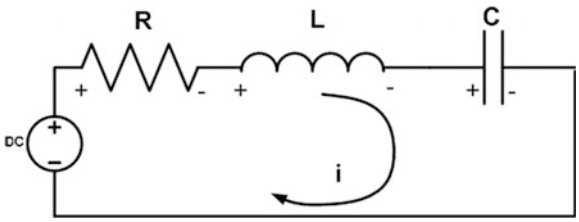

Fig. 5.6 RLC series circuit with constant voltage

Definition 16 (*RLC Series Circuit Model*)

```
⊢∀ R L C V Vo i t.rlc_ckt R L C V Vo i t =
  kvl [resistor_voltage R i;
       inductor_voltage L i;
       capacitor_voltage C i Vo; (λt. -V)] t
```

The list input of the function `kvl` is composed of all the elements of the circuit that have a voltage drop. The dc voltage source V is modeled in this list as a time-independent constant. The next step in the proposed methodology is to obtain a differential equation representation of the given AMS circuit. We formally verified this relationship as follows [31].

Theorem 2 *Differential Equation for the RLC Circuit*

```
⊢ ∀ R L C V Vo i t y.
(0 < y) ∧ (y < t) ∧
(∀x. 0≤x ∧ x≤t ⇒ i differentiable x) ∧
(∀x. 0≤x ∧ x≤t ⇒ ((λt.deriv i t))
                      differentiable x) ∧
(rlc_ckt R L C V Vo i t) ⇒
  (diff_eq_lhs(const_list[1/C;R;L]) i y = 0)
```

The conclusion of Theorem 2 describes the second-order differential equation corresponding to the RLC circuit given in the assumption using the function `rlc_ckt`. The theorem is verified under the assumptions that both the current function i and its first derivative are differentiable. It is also important to note that the theorem is valid for all time y in the interval $(0, t)$, where t represents the upper bound of the time for which the behavior of the function `rlc_ckt` is valid. Theorem 2 has been primarily verified using Theorem 1, some real analysis-based reasoning.

5.4.3.2 Delta-Sigma Modulator

In order to illustrate the proposed methodology, we present the formal verification of the first-order delta-sigma modulator, shown in Fig. 5.7, which is the widely used benchmark in formal verification of analog circuits.

The implementation model of this circuit can be obtained by applying KCL function at the input node of the op-amp:

Definition 17 (*Implementation Model of Delta-Sigma Modulator*)

```
⊢ ∀ R C Vin Vout Vc Veq y.
delta_sigma_imp R C Vin Vout Vc Veq y =
  (kcl [resistor_current R Vin;
        resistor_current R Vout;
        capacitor_current C (λx. -Vc x)] t)∧
 (Vout = (λt. Veq t - Vc t))
```

The next step is to formalize its specification:

Definition 18 (*Behavioral Model of Delta-Sigma Modulator*)

```
⊢ ∀ R C Vin Vout Veq y.
delta_sigma_behav R C Vin Vout Veq y =
  (diff_eq [1; R * C] Vout y =
  -Vin y + diff_eq [0; R * C] Veq y)
```

The function `diff_eq` accepts the list of coefficients of a differential equation, the differentiable function, and the differentiation variable and returns the corresponding differential equation.

Next, we formally verified the following implication between the implementation and specification of the given first-order delta-sigma modulator.

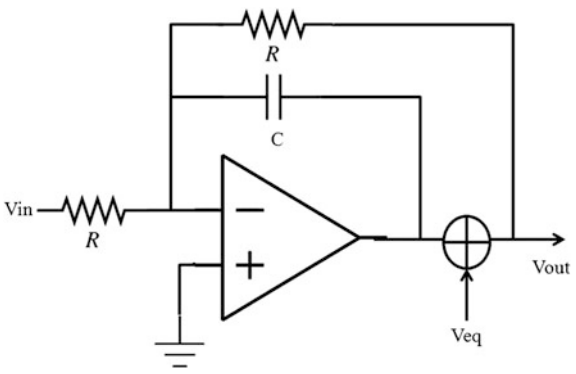

Fig. 5.7 First-order delta-sigma modulator

Theorem 3 *Implementation implies Specification*

```
⊢ ∀ R C Vin Vc Vout Veq t .
  delta_sigma_imp R C Vin Vc Vout Veq t⇒
  delta_sigma_behav R C Vin Vout Veq t
```

The proof was very straightforward due to the available formally verified properties and simplifiers for real analysis-related reasoning in HOL. The differential equation of Definition 18 does not have a closed-form mathematical solution, and thus, we feed it to a computer algebra system to obtain its solution and thus other interesting characteristics of the delta-sigma modulator.

The proof scripts for both of the application theorems are composed of just 300 lines approximately. This is far less than the proof script for the formalization, presented in the previous two subsections, which is more than 3500 lines of HOL code. This fact clearly indicates the usefulness of our foundational formalization associated with the proposed methodology. Just like the case studies, presented in this section, our formalization results can be utilized to automatically verify interesting properties of a wide variety of analog circuits in a straightforward manner and the results would be guaranteed to be correct due to the inherent soundness of theorem proving.

5.5 Summary

Early uncovering of design flows is a daunting procedure during the integration of digital and AMS components. The heterogeneous verification of AMS designs poses great challenges for the development of System-on-Chip because of the infinite state space composed of continuous and discrete states. In this chapter, we have presented two complementary formal verification methodologies that address this obstacle. The rigorous characteristics of the methodology strengthen the verification and provide a support for simulation through state space exploration and corner cases identification. Experimental results have proven the feasibility of the approach. The symbolic-based method can find application along the design flow of complex AMS designs. Formal verification can be applied to check conformance of reduced order models. We are currently expanding the application of formal verification as a guidance during circuit sizing. In addition, our formally verified exact solutions of differential equations can also be used to formally verify error bounds for the numerical method-based solutions for the analog circuits for which the differential equations do not have closed-form mathematical solutions. To broaden the scope of analog circuit verification, we also plan to extend the library of analog circuit components with diodes and transistors, etc. We are also working on developing reasoning support for non-homogeneous linear differential equations.

Finally, the calculus theories available in HOL-Light [32] are based on multivariate real numbers and thus can model complex numbers. Moreover, this work has been recently extended to formalize some Laplace transform theory [33]. Our formalization can be ported in a very straightforward manner to HOL-Light to be able to benefit from these mathematical foundations, which would enable handling the formal analysis of analog circuits in the complex plane.

References

1. Gielen, G.G., Rutenbar, R.A.: Computer-aided design of analog and mixed-signal integrated circuits. Proc. IEEE **88**(12), 1825–1852 (2000)
2. Shi, G., Tan, S.X.-D., Tlelo-Cuautle, E.: Advanced Symbolic Analysis for VLSI Systems. Springer, Berlin (2014) (ISBN 978-1-4939-1102-8)
3. Fakhfakh, M., Tlelo-Cuautle, E., Fernandez, F.V.: Design of Analog Circuits through Symbolic Analysis, 491 pages. Bentham Sciences Publishers Ltd., Sharjah (2012) (ISBN: 978-1-60805-425-1)
4. Johns, D., Martin, K.: Analog Integrated Circuit Design. Wiley, New York City (1997)
5. Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 118–149 (2003)
6. Moore, R.E.: Methods and Applications of Interval Analysis. Society for Industrial and Applied Mathematics, Philadelphia (1979)
7. Zaki, M.H., Tahar, S., Bois, G.: Formal verification of analog and mixed signal designs: a survey. Microelectron. J. **39**(12):1395–1404 (2008)
8. Kurshan, R.P., McMillan, K.L.: Analysis of digital circuits through symbolic reduction. IEEE Trans. Comput. Aided Des. **10**(11), 1356–1371 (1991)
9. Hartong, W., Klausen, R., Hedrich, L.: Formal verification for nonlinear analog systems: approaches to model and equivalence checking. In: Advanced Formal Verification, pp. 205–245. Kluwer, The Netherlands (2004)
10. Greenstreet, M.R., Mitchell, I.: Reachability analysis using polygonal projections. In: Hybrid Systems: Computation and Control, LNCS, vol. 1569, pp. 103–116, Springer, Berlin (1999)
11. Yan, C., Greenstreet, M.R.: Verifying an arbiter circuit. IEEE Form. Methods Comput. Aided Des. 1–9 (2008)
12. Zaki, M.H., Al Sammane, G., Tahar, S., Bois, G.: Combining symbolic simulation and interval arithmetic for the verification of AMS designs. IEEE Form. Methods Comput. Aided Des. 207–215 (2007)
13. Walter, D., Little, S., Myers, C.: Bounded model checking of analog and mixed-signal circuits using an SMT solver. In: Automated Technology for Verification and Analysis, LNCS, vol. 4762, pp. 66–81. Springer, Berlin (2007)
14. Dang, T., Donze, A., Maler, O.: Verification of analog and mixed-signal circuits using hybrid system techniques. In: Formal Methods in Computer-Aided Design, LNCS, vol. 3312, pp. 14–17. Springer, Berlin (2004)
15. Gupta, S., Krogh, B.H., Rutenbar, R.A.: Towards formal verification of analog designs. In: IEEE/ACM International Conference on Computer Aided Design, pp. 210–217 (2004)
16. G. Frehse, B.H. Krogh, R.A. Rutenbar. Verifying analog oscillator circuits using forward/backward abstraction refinement. IEEE/ACM Des. Autom. Test Eur. pp. 257–262 (2006)
17. Lata, K., Roy, S.K.: Formal verification of analog and mixed signal designs using SPICE circuit simulation traces. J. Electron. Test. **29**(5), 715–740 (2013)

18. Al-Sammane, G.: Simulation Symbolique des Circuits Decrits au Niveau Algorithmique. PhD thesis, Université Joseph Fourier, Grenoble, France (2005)
19. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
20. Al Sammane, G., Zaki, M., Tahar,S.: A symbolic methodology for the verification of analog and mixed signal designs. IEEE/ACM Des. Autom. Test Eur. 249–254 (2007)
21. Moore, J.S.: Introduction to the OBDD algorithm for the ATP community. J. Autom. Reason. 12(1):33–45 (1994)
22. Amla, N., Du, X., Kuehlmann, A., Kurshan, R.P., McMillan, K.L.: An analysis of SAT-based model checking techniques in an industrial environment. In: Correct Hardware Design and Verification Methods, LNCS, vol. 3725, pp. 254–268. Springer, Berlin (2005)
23. Wolfram, S.: Mathematica: A System for Doing Mathematics by Computer. Addison Wesley Longman Publishing, USA (1991)
24. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: Spaceex: scalable verification of hybrid systems. In: Computer Aided Verification, LNCS, vol. 6806, pp. 379–395. Springer, Berlin (2011)
25. Denman, W., Akbarpour, B., Tahar, S., Zaki, M., Paulson, L.C.P: Formal verification of analog designs using MetiTarski. In: Formal Methods in Computer Aided Design, pp. 93–100. IEEE, New York (2009)
26. Hanna, K.: Reasoning about Real Circuits, vol. 859, pp. 235–253. Springer, Berlin (1994)
27. Ghosh, A., Vemuri, R.: Formal verification of synthesized analog circuits. In: ACM/IEE International Conference on Computer Design, vol. 31, pp. 40–45 (1999)
28. Hanna, K.: Reasoning about analog level implementation of digital systems. Form. Methods Syst. Des. **16**(2), 127–158 (2000)
29. Taqdees, S.H., Hasan, O.: Formal verification of continuous models of analog circuits. In: Frontiers in Analog CAD, Poster Paper (2013)
30. Harrison, J.: Theorem Proving with the Real Numbers. Springer, Berlin (1998)
31. Usman, M., Hasan, O.: Formal verification of cyber-physical systems: coping with continuous elements. In: Computational Science and Its Applications, LNCS-Part 1, vol. 7971, pp. 358–371. Springer, Berlin (2013)
32. Harrison, J.: A HOL theory of Euclidean space. In: Theorem Proving in Higher Order Logics, LNCS, vol. 3603, pp. 114–129. Springer, Berlin (2005)
33. Taqdees, S.H., Hasan, O.: Formalization of Laplace transform using the multivariate calculus theory of HOL-light. In: Logic for Programing Artificial Intelligence and Reasoning, LNCS, vol. 8312, pp. 744–758. Springer, Berlin (2013)