Integrating SMT with Theorem Proving for Analog/Mixed-Signal Circuit Verification[⋆]

Yan Peng and Mark Greenstreet

University of British Columbia
Vancouver, British Columbia, Canada
{yanpeng,mrg}@cs.ubc.ca

Abstract. We present our integration of the Z3 SMT solver into the ACL2 theorem prover and its application to formal verification of analog-mixed signal circuits by proving global convergence for a state-of-the-art digital phase-locked loop (PLL). SMT (satisfiability modulo theory) solvers eliminate much of the tedium associated with detailed proofs by providing automatic reasoning about propositional formulas including equalities and inequalities of polynomial functions. A theorem prover complements the SMT solver by providing a proof structuring and proof by induction. We use this combined tool to show global convergence (i.e. correct start-up and mode-switching) of a digital PLL. The PLL is an example of a second-order hybrid control system; its verification demonstrates how these methods can address challenges that arise when verifying such designs.

1 Introduction

We present our integration of the Z3 SMT solver [1] into the ACL2 theorem prover [2]. With this approach, high-level proof structure and proof techniques such as induction can be handled by the theorem prover while many tedious details for verifying real-world designs discharged by the SMT solver. The implementation presented in this paper supports booleans, integers, and rationals/reals, and our approach could be readily extended to other types including arrays, lists, strings, and more general algebraic data types. For soundness, we want to rely on ACL2, Z3, and as little other code as possible. We also want Z3 to be easily used from within ACL2; thus, our interface performs many, automatic transformations of ACL2 formulas to convert them into the restricted form required by Z3. We resolve these seemingly conflicting objectives with a software architecture that divides the ACL2-to-Z3 translation process into two phases: most of the transformations are performed in the first phase, and the result is verified by ACL2. The second phase is a very simple direct translation from the s-expressions of ACL2 into their counterparts for Z3's Python API.

We demonstrate our approach by using it to verify global convergence for an all-digital phase-locked loop (PLL). The PLL is an example of an analog-mixed signal (AMS) design. The AMS approach has emerged as the dominant paradigm for implementing analog operations where digital circuits replace many analog functions and

[⋆] This research was supported by grants from NSERC Canada and Intel.

provide adaptation functions to compensate for variations in process, device, and operating conditions. AMS designs play an important role in nearly every computing, communication, and cyber-physical systems. These designs pose serious simulation challenges because they involve an extremely wide range of time-scales from sub-picosecond (e.g. for oscillator jitter) to milliseconds or longer for software controlled adaptation loops or interactions with mechanical sensors and systems. Formal methods can verify properties that are intractable or impossible to show with simulation. We consider global convergence: showing that an AMS circuit converges to the intended operating mode from all initial conditions. This requires modeling the large-scale, non-linear behavior of the analog components. If such non-linearities create an unintended basin of attraction, then the AMS circuit may fail to converge to the intended operating point. AMS circuits may make many mode changes per second to minimize power consumption, adapt to changing loads, or changes in operating conditions. Each of these mode changes requires the AMS circuit to converge to a new operating region. Once the AMS circuit is in the small operating region intended by the designer, small-signal analysis based on linear-systems theory is sufficient to show correct operation [3, 4].

The key contributions of this paper are:

- A demonstration that the arithmetic decision procedures of an SMT solver can be exploited effectively when verifying properties of physical systems with continuous models.
- A description of the challenges that arise when using a SMT solver with a theorem prover and present our solutions to these issues.
- the first integration of an SMT solver into the ACL2 theorem prover.
- A model for a state-of-the art digital PLL with recurrences using rational functions that can be used for evaluating other verification approaches.
- A proof of global convergence of the digital PLL.

2 Related Work

There has been extensive work in the past decade on integrating SAT and SMT solvers into theorem provers including [5–11]. Many of these papers have followed Harrison and Théry's "skeptical" approach and focused on methods for verifying SMT results within the theorem prover using proof reconstruction, certificates, and similar methods. Several of the papers showed how their methods could be used for the verification of concurrent algorithms such as clock synchronization [6], and the Bakery and Memoir algorithms [9]. While [6] used the CVC-Lite [12] SMT solver to verify properties of simple quadratic inequalites, the use of SMT in theorem provers has generally made light use of the arithmetic capability of such solvers. In fact [10] reported *better* results for SMT for several sets of benchmarks when the arithmetic theory solvers were disabled!

The work that may be the most similar to ours is [11] that presents a translation of Event-B sequents from Rodin [13] to the SMT-LIB format [14]. Like our work, [11] verifies a claim by using a SMT solver to show that its negation is unsatisfiable. They address issues of types and functions. They perform extensive rewriting using Event-B sequents, and then have simple translations of the rewritten form into SMT-LIB. While noting that proof reconstruction is possible in principle, they do not appear to implement such measures. The main focus of [11] is supporting the set-theoretic constructs of Event-B. In contrast, our work shows how the procedures for non-linear arithmetic of a modern SMT solver can be used when reasoning about VLSI circuits.

Our work demonstrates the value of theorem proving combined with SMT solvers for verifying properties that are characterized by functions on real numbers and vector fields. Accordingly, the linear- and non-linear arithmetic theory solvers have a central role. As our concern is bringing these techniques to new problem domains, we deliberately take a pragmatic approach to integration and trust both the theorem prover and the SMT solver.

Prior work on using theorem proving methods to reason about dynamical systems includes [15] which uses the Isabelle theorem prover to verify bounds on solutions to simple ODEs from a single initial condition. In contrast, we verify properties that hold from *all* initial conditions. Harutunian [16] presented a very general framework for reasoning about hybrid systems using ACL2 and demonstrated the approach with some very simple examples. Here we demonstrate that by discharging arithmetic proof obligations using a SMT solver, it is practical to reason about much realistic designs.

The past decade has also seen a rapidly growing interest in applying formal methods to analog and mixed-signal designs. Much of this work goes back to early model-checking results by Kurshan and MacMillan [17]. Other early work includes [18–21]. To verify phase-locked loops, Dong *et al.* [22] proposed using property checking for AMS verification, including PLLs. Shortly after the work by Dong *et al.*, Jesser and Hedrich [23] described a model-checking result for a simple analog PLL. Althoff *et al.* [24] presented the verification of a charge-pump PLL using an approach that they refer to as "continuization." They use a purely linear model for the components of their PLL, and their focus is on the switching activities of the phase-frequency detector, in particular, uncertainties in switching delays.

More recently, Lin et al. [25, 26] developed an approach for verifying a digital PLL using SMT techniques. To the best of our knowledge, they are the first to claim formal verification of a digital PLL. They consider a purely linear, analog model and then reason about the discrepancies between this idealized model and a digital implementation. They use the KRR SMT solver to verify bounds on this discrepancy. They verify bounds on the lock time of a digitally intensive PLL assuming that most of the digital variables are initialized to fixed values, and that only the oscillator phase is unknown. Our work shows initialization for a different PLL design over the complete state space.

Using the SpaceEx [27] reachability tool, Wei *et al.* [28] presented a verification of the same digital PLL as described in this paper. That work made a over-approximation of the reachable space by over-approximating the recurrences of the digital PLL with linear, differential inclusions. As SpaceEx could not verify convergence property for the entire space in a single run, [28] broke the problem into a collection of lemmas that were composed manually. Their work demonstrated the need for some kind of theorem-proving tool to compose results. Furthermore, they could not show the limit cycles that our proof does; therefore their proof does not provide as tight of bounds on PLL jitter and other properties as can be obtained with our techniques.

3 Integrating Z3 into ACL2

Theorem provers and SMT solvers provide complementary reasoning capabilities. The main challenge is connecting the two with an interface that is both useful and trustworthy. We achieve these goals by using a two-step translation process. The first step translates the user-provided goal into a minimal subset of operations that all have direct counterparts in the logic of the SMT solver. The second step performs a direct translation of this expanded and simplified version of the goal into the syntax and logic of the SMT solver. The SMT solver then proves the goal, provides a counter-example, or reports that it could not decide. A key feature of this architecture is that most of the complexity of the translation process is in the first step. The first step translates a goal in the theorem prover logic to an equivalent or stronger goal, still in the theorem prover logic. We use the theorem prover to verify this implication for each translated goal. Thus, the first step does not introduce any soundness assumptions beyond our existing faith in the theorem prover. The second step provides the translation between the theorem prover and SMT solver. This is trusted code, and our design allows it to be very simple and easily reviewed by other humans.

This section describes our solution to integrating Z3 into ACL2 in more detail. We describe the issues that arose to ensure the soundness of our implementation and features that we've included to make the combination easy to use. First, Section 3.1 gives a very brief description of the key features of ACL2 and Z3 used in our design.

3.1 ACL2 and Z3

ACL2 [2] is a theorem prover for programs written in a comprehensive, applicative subset of Common Lisp. ACL2 provides a very general notion of induction based on recursive function definition in lisp: every recursive function defines a corresponding induction schema. This is ideal for our application where we are using a theorem prover to compensate for the lack of induction capabilities in SMT solvers. While ACL2 supports many other methods for proving goals in its "waterfall", our approach is to allow ACL2 to automate a proof where it can, and use an integrated SMT solver to discharge tedious obligations, especially those involving systems of non-linear inequalities that frequently arise when reasoning about AMS circuits or other physical systems. ACL2 supports the integration of external decision procedures through its clause processor mechanism described below.

ACL2 represents a proof goal as a conjunction of clauses, where each clause is a disjunction of terms. The terms can be arbitrary s-expressions. ACL2 supports the inclusion of user-defined "clause processors". A clause processor takes an ACL2 clause (i.e. proposition) as an argument and returns a list of clauses with the interpretation that the conjunction of the result clauses implies the original clause. In particular, if the result list is empty, then the clause processor is asserting that the original clause is true for all valuations of any free variables. ACL2 supports two types of clause processors: verified and trusted. A *verified* clause processor is written in the ACL2 subset of Common Lisp and proven correct by ACL2. A *trusted* clause processor does not require a correctness proof; instead, all theorems are tagged to identify the trusted processors that they may depend on. The soundness of the trusted clause processor becomes, in effect,

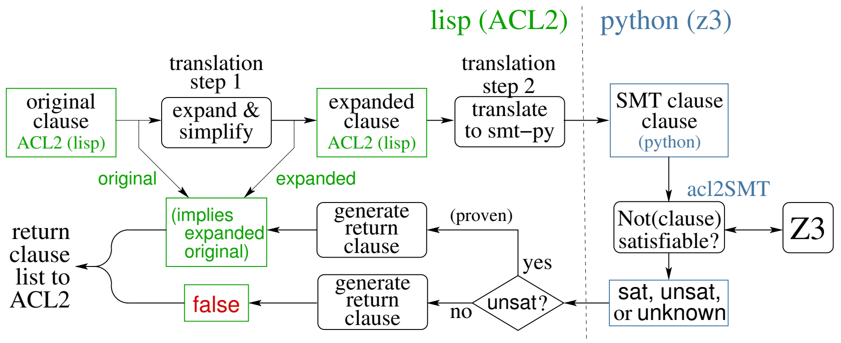


Fig. 1. The Clause Processor

a hypothesis of any theorem that uses the clause processor. We connected the Z3 SMT solver to ACL2 by writing a trusted clause processor.

Z3 [1] is a SMT solver developed at Microsoft Research that has been used in many software verification tools. Z3 combines procedures for boolean combinations of linear equalities and inequalities over linear, polynomial, and rational functions along with and operations on arrays and algebraic data types within the framework of a CDCL satisfiability solver [29]. We make incidental use of some of the other capabilities of Z3 such as lists to simplify the implementation of our interface. Formulas for Z3 can be written using the SMTLIB format [14], an Ocaml API, or a Python API (z3py). We used z3py for the ease of prototyping in Python.

3.2 The Clause Processor

Figure 1 shows the architecture of our clause processor. The critical part of this design for soundness is the second translation step. This step supports a very small subset of the ACL2 logic consisting of nineteen functions:

- five arithmetic functions – binary `plus`, `minus`, and `times`; and unary `negation`, and `reciprocal`.
- five comparison functions – `equal`, $<$, $\leq$, $\geq$, and $>$.
- three more logical operations – `if`, `not`, and `implies`.
- three functions for declaring the types of Z3 variables – `booleanp`, `integerp`, and `rationalp`.
- three functions to support other ACL2 constructions – `array`, `lambda`, and `nth`.

This translation step is implemented using 220 lines of lisp code (translate step 2 in the figure), and 130 lines of python (class `acl2SMT` in the figure). The lisp code is driven by an association list to map lisp functions to their python counterparts. The python code defines an object whose methods call the appropriate Z3 functions. As classes implementing the same interface could be defined for other SMT solvers, our architecture is largely solver agnostic. The code is straightforward and easily inspected.

In principle, a user could transform a more general ACL2 clause into one that uses the minimal set of operators described above by guiding ACL2's rewriting process. The tedious effort of doing so would largely nullify the advantages promised by using an external decision procedure. To make the SMT solver of practical use, the first step of the translation process converts a richer subset of the ACL2 logic into the minimal form accepted by the final translation. As described below, the user can provide hints that produce a translated clause that is *stronger* than the original clause. Further strengthening of the clause can occur because of the way we handle the connection between ACL2's use of rational numbers and the use of real numbers in the logic of Z3.

Once the clause has been translated into the logic of the SMT solver, we ask the SMT solver to prove it by showing that the negation of the clause is unsatisfiable. If the expanded clause is proven to be a theorem by the SMT solver, then the clause processor returns `(implies expanded original)` to ACL2. In other words, the original clause is a theorem if ACL2 can show that the expanded clause implies the original. The soundness of the connection to the SMT solver does not depend on the first step of the translation. If the SMT solver finds a satisfying assignment to the negated clause, the original claim *might* not be a theorem. Presently, we report the failure, but do not provide any proposed counter-example to ACL2.

Given this framework, the remainder of this section describes the logical issues that arose when integrating Z3 into ACL2 and presents our solutions.

3.3 Connecting the logics

The logic of ACL2 is much more expressive than that of SMT solvers such as Z3. In particular,

- ACL2 is untyped, but Z3 requires a declared sort (i.e. type) for each variable.
- ACL2 supports rational numbers, but Z3 uses reals.
- Users of ACL2 usually make extensive use of user-defined, recursive functions. Z3 only provides uninterpreted functions.
- The antecedent of an ACL2 clause may imply other facts that can be proven by ACL2 and are needed by the SMT solver but that cannot be derived by the SMT solver.
- Substitutions: a clause may include (in)equalities over terms that are neither polynomials nor rational functions and thus outside the domain of Z3's non-linear solver.
- Hints: our clause processor may return clauses for ancillary conditions that ACL2 cannot discharge without further guidance.

We show in the remainder of this section how each of these issues are naturally addressed within the clause processor architecture described above.

Typed vs. untyped logics: ACL2 is untyped and all functions are total. However, ACL2 does provide type predicates, and it is common to write theorems of the form:

```
(thm (implies (and (and type−assertion_1 type−assertion_2 ...)
                   (and other-hypotheses))
              conclusion))
```

where *type-assertions* are of the form `(rationalp x)`, `(integerp n)`, etc. We simply require that theorems be stated with this structure. Our clause processor checks for this structure, identifies the type assertions, and generates the corresponding variable declarations for the SMT solver. This is done in the second translation step, and soundness requires *exact* correspondence between the ACL2 and Z3 types. This is the case for booleans and integers (e.g. ACL2 and Z3 have the same definition of integer additions, etc.). However, it raises an issue for ACL2 rationals vs. Z3 reals which we discuss next.

Real numbers and rationals: Our translator represents variables that are restricted to rational number values in ACL2 to ones that are real valued in Z3. We need to ensure that results from Z3 are sound in ACL2. For example, in ACL2 can prove that there is no (rational) number x such that $x^2 = 2$. On the other hand, Z3 can prove that there is a (real) number x such that $x^2 = 2$. To preserve soundness, we restrict the use of the SMT solver to discharging clauses where all variables are universally quantified, noting that

$$\forall x \in \mathbb{R}.\, p(x) \;\Rightarrow\; \forall x \in \mathbb{Q}.\, p(x)$$

because $\mathbb{Q} \subset \mathbb{R}$. In practice, this is not a serious restriction.

Another consequence of this difference in representation is that Z3 may produce a counter-example where some variables are assigned irrational (in particular, algebraic) values. As illustrated by the $x^2 = 2$ example above, such a counter-example is not valid in ACL2. When Z3 produces a model for a formula, each value can be identified as being integer, rational, or algebraic. Thus, we could check if a counter-example generated by Z3 could be used in ACL2. We have not needed existential witnesses, and restricting the SMT solver as described above has been completely satisfactory.

An alternative would be to use ACL2r [30]. We have used our clause processor with both ACL2 and ACL2r, and the same proof of convergence for the digital PLL works in both systems. Presently, ACL2r is not fully upward compatible with ACL2, and we chose to start with ACL2 because of its more comprehensive environment for proof development.

Another difference between ACL2 and Z3 for numerical problems is division by zero. While ACL2 and Z3 require all operations to be total, ACL2 defines `(/ x 0)` to be `0` for all `x`; where as Z3 considers the quotient to be an unspecified number. Our solution is implement reciprocal in the acl2SMT module to match the ACL2 definition:

```
    def reciprocal(self, x): self.ifx(x == 0.0, 0.0, 1.0/x)
```

Functions. ACL2 users naturally use the expressiveness of lisp to write concise theorem statements. This includes user-defined and library functions. While Z3 and other SMT solvers support uninterpreted functions, this is almost always too imprecise to prove real theorems. Many functions that appear in proofs are non-recursive; in other words, they are macros. These are easily eliminated by expanding the function. Of course, clauses can also include recursive functions.

Our clause processor accepts a user-provided "hint" called `:functions` that says what functions should be expanded, to what depth they should be expanded, and the type of the value produced by the function. These hints are used in the first translation step to expand function calls to the specified depth. Any deeper calls are replaced by an unconstrained variable of the specified return type. These transformations are performed in the first translation step; therefore, ACL2 verifies the soundness of this transformation.

The expansion of function calls replaces `(f actual1 actual2 ...)` with

```
   ( (lambda (newVar1, newVar2, newVar3) rewritten-body-of-f)
     (actual1 actual2 ...))
```

where the body of `f` is rewritten by replacing formal parameters with the fresh variables (i.e. names that don't otherwise appear in the clause) and recursively expanding any function calls in the body. The second step translates this into the corresponding Python lambda expression. The `acl2SMT` object provides a method that creates names for the SMT solver for fresh variables.

This mechanism for supporting functions is also used to support `let` operations in the user-provided clause – in fact, the ACL2 macro expansion turns them into the form described above without any assistance from our clause processor.

Adding Hypotheses: The hypotheses of a clause may imply other facts that are not derivable by the SMT solver but that can be readily shown within the theorem prover. These include previously established theorems and claims that have straightforward induction proofs. The SMT solver may need these additional facts to prove a clause.

Our clause processor accepts a user-provided hint called `:hypothesize` to add additional hypotheses to the clause. The hypotheses are added in the first translation step; therefore, ACL2 verifies the soundness of this transformation.

Non-polynomials: the non-linear arithmetic procedures in Z3 only support polynomials and rational functions. For example, our reasoning about the digital PLL uses the Common Lisp function `(expt r n)` which computes r^n for integer n. Our clause processor accepts a user-provided hint called `:let` to replace all occurrences of given subexpression with a new unconstrained variable of the user specified type. Our clause processor then adds a proof obligation (to be returned to ACL2) that the type of the subexpression always matches the user specified type. These substitutions are performed in the first translation step; therefore, ACL2 verifies the soundness of this transformation. Typically, the user will also use `:hypothesize` hints to state constraints that must hold for the value of this new variable. For example, we might include the hint

```
  :hypothesize (equal (expt x (+ n 2)) (* x x (expt x n)))
```

As noted above, the soundness of these added hypotheses are verified by ACL2.

Nested Hints: as described above, our clause processor returns a new clause to check the soundness of the translation and any user-provided assertions. In practice, this means that Z3 establishes the truth of a complicated system of equalities and inequalities, and ACL2 is required to discharge a handful of much simpler conditions. Occasionally, these returned clauses can be non-trivial, and our clause processor accepts a user-provided hint called `:use` that allows the user to attach hints to the clauses returned to ACL2. A common form of this is to prove a lemma within ACL2 that corresponds to a `:hypothesize` hint. The user may then employ a `:use` hint to tell ACL2 how to instantiate the lemma to discharge the added hypothesis. Proof hints in ACL2 do not change the meaning of the formula that ACL2 is attempting to prove; they only guide the theorem prover to use proof methods that the user believes will succeed. Thus, these hints do not affect the soundness of our implementation.

3.4 Soundness of the connection

The previous section described the main issues that arose when integrating an SMT solver into the ACL2 theorem prover. The two-step translation process allows the more complicated transformations to be performed in the first step, and the result is verified by ACL2. The second step is very simple, and that is the only part that is trusted for soundness. This section summarizes how a clause is discharged using our clause processor by showing the logical transformations that are performed at each step.

Originally, the clause processor is asked to prove

$$(\mathit{antecedent} \wedge \mathit{typePredicates}) \;\rightarrow\; \mathit{claim} \tag{1}$$

Let *Goal* denote this proposition. Furthermore, the user may provide the clause processor with various hints about function expansion, subexpression substitution, additional hypotheses, and hints to give back to ACL2. The first translation step of the clause processor rewrites *antecedent* to *antecedent′* and *claim* to *claim′*. It also introduces new hypotheses, both from `:hypothesize` hints and the type assumptions for functions and `:let` hints. Let *moreHyps* denote these added hypotheses. The SMT solver attempts to prove:

$$(\mathit{antecedent}' \wedge \mathit{typePredicates} \wedge \mathit{moreHyps}) \;\rightarrow\; \mathit{claim}' \tag{2}$$

Let *Goal′* denote this "expanded" proposition. If the SMT solver proves *Goal′*, then the clause processor returns the following clauses to ACL2:

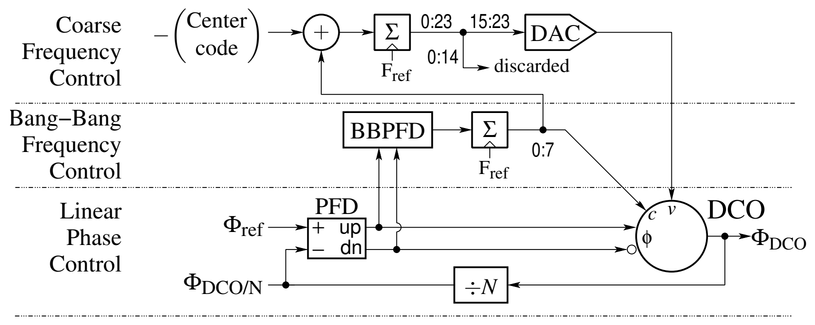
$$\begin{gathered} \mathit{Goal}' \rightarrow \mathit{Goal} \\ (\mathit{antecedent} \wedge \mathit{typePredicates}) \rightarrow \mathit{moreHyps} \end{gathered} \tag{3}$$

Where the last line is one clause for each added hypothesis in the actual implementation. The original claim (Eq. 1) is an immediate consequence of the clause established by the SMT solver (Eq. 2) and the clauses subsequently discharged by ACL2 (Eq. 3).

4 Verifying a Digital PLL

Figure 2 shows the digital phase-locked-loop (PLL) verified in this paper; it is a simplified version of the design presented in [31]. The purpose of this PLL is to adjust the digitally-controlled oscillator (DCO) so that its output, Φ_{DCO} has a frequency that is N times that of the reference input, Φ_{ref} and so that their phase match (i.e. each rising edge of Φ_{ref} coincides with a rising edge of Φ_{DCO}). The three control-paths shown in the figure make this a third-order digital control system. By design, the lower two paths dominate the dynamics making the system effectively second-order.

The DCO has three control inputs: ϕ, c, and v. The ϕ input is used by a proportional control path: if Φ_{ref} leads $\Phi_{DCO/N}$ then the PFD will assert `up`, and the DCO will run faster for a time interval corresponding to the phase difference. Conversely, if Φ_{ref} lags $\Phi_{DCO/N}$, the `dn` signal will be asserted, and the DCO will run slower for a time interval corresponding to the phase difference. If the frequencies of Φ_{ref} and $\Phi_{DCO/N}$ are not



Φ_{ref} is the reference signal whose frequency is denoted by f_{ref}.
Φ_{DCO} is the output of the digitally controlled oscillator whose frequency is denoted by f_{DCO}.
Labels of the form lo:hi denote bits lo through hi (inclusive) of a binary value.

Fig. 2. A Digital Phase-Locked Loop

closely matched, then the PFD simply outputs `up` (resp. `dn`) if the frequency of $\Phi_{DCO/N}$ is lower (resp. higher) than that of Φ_{ref}.

The c input of the DCO is used by the integral control path. The DCO in [31] is a ring-oscillator, and the c input controls switched capacitor loads on the oscillator – increasing the capacitive load decreases the oscillator frequency. The bang-bang phase-frequency detector (BBPFD) controls whether this capacitance is increased one step or decreased one step with for each cycle of Φ_{ref}. The c input provides a fast tracking loop.

The v input of the DCO is used to re-center c to restore tracking range. This input sets the operating voltage of the oscillator – the oscillator frequency increases with increasing v. The accumulator for this path is driven by the difference between c and its target value c_{center}.

As a control system, the PLL converges to a switching surface where c and ϕ fluctuate near their ideal values. As presented in [31] these limit-cycle variations are designed to be slightly smaller than the unavoidable thermal and shot-noise of the oscillator. Furthermore, the time constants of the three control loops are widely separated. This facilitates intuitive reasoning about the system one loop at a time – it also introduces stiffness into the dynamics that must be considered by any simulation or reachability analysis. We believe that these characteristics of convergence to a switching surface and stiffness from multiple control loops with widely separated tracking rates are common in digitally controlled physical systems. This motivates using the digital PLL as a verification example and challenge.

4.1 Modeling the Digital PLL

From Spectre simulations, we observe that the oscillator frequency is very nearly linear in v and nearly proportional to the inverse of c for a wide range of each of these parameters. The phase error, ϕ is a continuous quantity, but the values of c and v are determined by the digital accumulators that are updated on each cycle of the reference clock, f_{ref}. This motivates modeling the PLL using a discrete-time recurrence for real-valued variables:

$$\begin{aligned}
c(i+1) &= \min(\max(c(i)+g_c\,\mathrm{sgn}(\phi), c_{\min}), c_{\max})\\
v(i+1) &= \min(\max(v(i)+g_v(c_{center}-c(i)), v_{\min}), v_{\max})\\
\phi(i+1) &= \mathrm{wrap}(\phi(i)+(f_{DCO}(c(i),v(i))-f_{ref})-g_\phi\,\phi(i))\\
f_{DCO}(c,v) &= \tfrac{1+\alpha v}{1+\beta c} f_0 \qquad\qquad (4)\\
\mathrm{wrap}(\phi) &= \mathrm{wrap}(\phi+1), \text{ if } \phi \le -1\\
&= \phi, \qquad\qquad\quad \text{if } -1 < \phi < 1\\
&= \mathrm{wrap}(\phi-1), \text{ if } 1 \le \phi
\end{aligned}$$

where g_c, g_v, and g_ϕ are the gain coefficients for the bang-bang frequency control, coarse frequency control, and linear phase paths respectively. The coefficient α is the slope of oscillator frequency with respect to v, and β is the slope of oscillator period with respect to c; both are determined from simulation data. We measure phase leads or lags in cycles: $\phi = 0.1$ means that $\Phi_{DCO/N}$ leads Φ_{ref} by 10% of the period of Φ_{ref}. We say that c is "saturated" if $(c = c_{\min}) \wedge (\phi < 0)$ or $(c = c_{\max}) \wedge (\phi > 0)$. Likewise, v is saturated if $(v = v_{\min}) \wedge (c > c_{center})$ or $(v = v_{\max}) \wedge (c < c_{center})$. In this paper, we scale f_{ref} to 1. With similar scaling, we choose $g_c = 1/3200$, $g_v = -gc/5$, and $g_\phi = 0.8$. We assume bounds for c of $c_{\min} = 0.9$ and $c_{\max} = 1.1$ with $c_{center} = 1$ and bounds for v of $v_{\min} = 0.2$ and $v_{\max} = 2.5$. With these parameters, the PLL is intended to converge to a small neighbourhood of $c = c_{center} = 1$; $v = f_{ref} c_{center} = 1$ and $\phi = 0$.

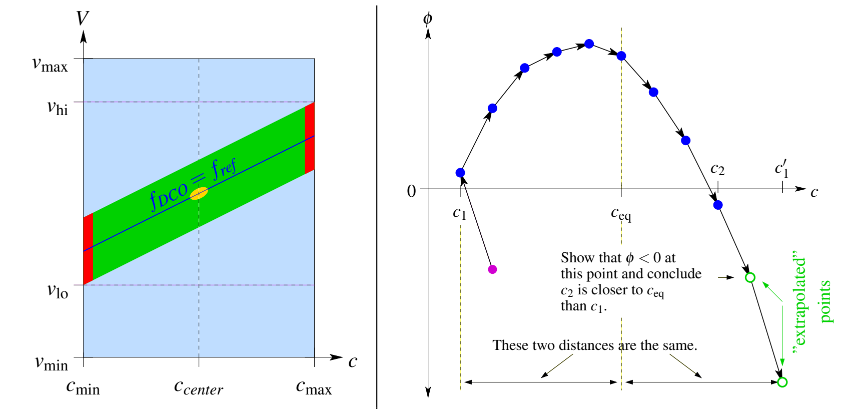
4.2 Proving Global Convergence

Our verification proceeds in three phases as depicted in Fig. 3.a. First we show that for all trajectories starting with $c \in [c_{\min}, c_{\max}]$, $v \in [v_{\min}, v_{\max}]$, and $\phi \in [-1, +1]$ (the blue regions 3.a) the trajectory eventually reaches a relatively narrow stripe (the red and green regions) for which $f_{DCO} \approx f_{ref}$. The proof is based on a simple ranking function that Z3 easily verifies. By proving this we've shown that the non-linearities of the global model do not create unintended stable modes.

The second part of the proof pertains to the small, red stripes where $f_{DCO} \approx f_{ref}$ but c is close enough to $c_{\min}$ or $c_{\max}$ that saturation remains a concern. Consider the red stripe near $c = c_{\min}$. We use Z3 as a bounded model checker to show that v increases, and that c "tracks" v to keep f_{DCO} close to f_{ref} and θ small. Together, these results show that all trajectories eventually enter the region shown in green in Fig. 3.

The final part of the proof shows convergence to the limit-cycle region, shown in yellow in Fig. 3.a. The key observation here is that ϕ repeatedly alternates between positive and negative values. For any given value of v, we can calculate the value of c for which $f_{DCO}(c, v) = f_{ref}$ — call this $c_{\mathrm{eq}}(v)$. Figure 3.b depicts a trajectory from a rising zero-crossing of ϕ to a falling crossing. Let c_1 be the value of c following a rising zero-crossing of ϕ, and let c_2 be the value of c at the subsequent falling crossing. We note that $c_1 < c_{\mathrm{eq}}(v) < c_2$.

The obvious way to show convergence is to show that c_2 is closer to c_{eq} than c_1 is. However, this involves calculating the recurrence step at which ϕ makes its falling crossing of zero, and that involves solving a non-linear system of equations. Although



(a) Coarse Convergence (b) Fine Convergence

Fig. 3. Global Convergence Proof

Z3 has a non-linear arithmetic solver, it does not support induction as would be required with an arbitrary choice for c_1. Instead, we extrapolate the sequence to the last point to the right of c_{eq} that is closer to c_{eq} than c_1 is. We use formula from Eq. 4 for computing $c(i+1)$ assuming that $\operatorname{sgn}\phi = 1$; either this assumption is valid for the whole sequence, or ϕ had a falling crossing even earlier. Either is sufficient to show convergence. The proof involves solving the recurrence, and rewriting the resulting formula. The key inequality has exponential terms of the form $(1-g_\phi)^n$ multiplied by rational function terms of the other model parameters. We use the substitution technique from Section 3.2 to replace these non-polynomial terms, and add a `:hypothesize` hint that $0 < (1-g_\phi)^n < 1$. ACL2 readily discharges this added hypothesis using a trivial induction.

Our proof is based on a 13-page, hand-written proof. The ACL2 version consists of 75 lemmas, 10 of which were discharged using the SMT solver. Of those ten, one was the key, polynomial inequality from the manual proof. The others discharged steps in the manual derivation that were not handled by the standard books of rewrite rules for ACL2. ACL2 completes the proof in a few minutes running on a laptop computer. We found one error in the process of transcribing the hand-written proof to ACL2.

We completed much of the proof using ACL2 alone while implementing the clause processor. We plan to rewrite the proof to take more advantage of the SMT solver and believe that the resulting proof will be simpler, focus more on the high-level issues, and be easer to write and understand. When faced with proving a complicated derivation, one can guide ACL2 through the steps of the derivation, or just check the relationship of the original formula to the final one using the SMT solver. The latter approach allows novice users (including the authors of this paper) to quickly discharge claims that would otherwise take a substantial amount of time even for an expert. As noted before, if Z3 finds a counter-example, we do not return it as a witness for ACL2. However, our clause processor prints the counter-example (in its Z3 representation) to the ACL2 proof log. The user can examine this counter-example; in practice, it often points directly to the problem that needs to be addressed.

The ACL2 formulation enabled making generalizations that we would not consider making to the manual proof. In particular, the manual proof assumed that $c_{\mathrm{eq}} - c_1$ was an integer multiple of g_1. After verifying the manual proof, we removed this restriction – this took about 12 hours of human time, most of which was to introduce an additional variable $0 \leq d_c < 1$ to account for the non-integer part. We also generalized the proof to allow v to an interval whose width is a small multiple of $|g_2(c_{\max} - c_{\min})|$. This did not require any new operators and took about 3 hours of human time. The interval can be anywhere in $[v_{\mathrm{lo}}, v_{\mathrm{hi}}]$. This shows that the convergence of c and ϕ continues to hold as v progresses toward $f_{ref}c_{center}$. It also sets the foundation for verifying the PLL with a more detailed model including the $\Delta\Sigma$ modulator in the c path, an additional low-pass filter in the v path, and adding error terms in the formula for $f_{DCO}(c, v)$.

5 Conclusions

This paper presented the integration of the Z3 SMT solver into the ACL2 theorem prover and demonstrated its application for the verification of global convergence for a digital PLL. The proof involves reasoning about systems of polynomial and rational function equalities and inequalities, which is greatly simplified by using Z3's non-linear arithmetic capabilities. ACL2 complements Z3 by providing a versatile induction capability along with a mature environment for proof development and structuring. Section 3 described technical issues that must be addressed to ensure the soundness, of the integrated prover, usability issues that are critical for the tool to be practical, and our solutions to these challenges.

Section 4 showed how this integrated prover can be used to verify global convergence for a digital phase-locked loop from all initial states to the final limit-cycle behaviours. The analysis of the limit cycle behaviour requires modeling the PLL with recurrences. Such limit cycles are not captured by continuous approximations used in [24, 28]. Our approach allowed uncertainty in the model parameters and not just in the signal values. The reachability tools cited above require fixed model parameters. Our approach shows much more promise for verification that accounts for device variability and other uncertainties.

Prior work on integrating SMT solvers into theorem provers has focused on using the non-numerical decision procedures of an SMT solver. Our work demonstrates the value of bringing an SMT solver into a theorem prover for reasoning about systems where a digital controller interacts with a continuous, analog, physical system. The analysis of such systems often involves long, tedious, and error-prone derivations that primarily use linear algebra and polynomials. We have shown that these are domains where SMT solvers augmented with induction and proof structuring have great promise. We are currently exploring using our methods to verify other AMS designs as well as to similar problems that arise in hybrid control systems and machine learning.

Acknowledgements

We thank Leo Moura, David Rager, Jijie Wei, and Ge Yu for helpful discussions about this research.

References

1. L. L. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*. Springer, 2008, pp. 337–340, LNCS 4963. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78800-3_24
2. M. Kaufmann, J. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.
3. K. S. Kundert, "Introduction to RF simulation and its application," *IEEE J. Solid-State Circuits*, vol. 34, no. 9, pp. 1298–1319, 1999. [Online]. Available: http://dx.doi.org/10.1109/4.782091
4. J. Kim, M. Jeeradit, B. Lim, and M. A. Horowitz, "Leveraging designer's intent: a path toward simpler analog CAD tools," in *Custom Integrated Circuits Conf.*, Sep. 2009, pp. 613–620. [Online]. Available: http://dx.doi.org/10.1109/CICC.2009.5280741
5. S. McLaughlin, C. Barrett, and Y. Ge, "Cooperating theorem provers: A case study combining HOL-Light and CVC Lite," in *3rd Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2005, pp. 43–51. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2005.12.005
6. P. Fontaine, J.-Y. Marion *et al.*, "Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants," in *TACAS*. Springer, 2006, pp. 167–181, LNCS 3920. [Online]. Available: http://dx.doi.org/10.1007/11691372_11
7. F. Besson, "Fast reflexive arithmetic tactics the linear case and beyond," in *Int'l. Workshop on Types for Proofs and Programs*. Springer, 2007, pp. 48–62, LNCS 4502. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74464-1_4
8. M. Armand, G. Faure *et al.*, "A modular integration of SAT/SMT solvers to Coq through proof witnesses," in *1st Int'l. Conf. Certified Programs and Proofs*, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25379-9_12
9. S. Merz and H. Vanzetto, "Automatic verification of TLA+ proof obligations with SMT solvers," in *18th LPAR*, Mar. 2012. [Online]. Available: https://hal.inria.fr/hal-00760570/document
10. J. C. Blanchette, S. Böhme, and L. C. Paulson, "Extending Sledgehammer with SMT solvers," *J. of Automated Reasoning*, vol. 51, no. 1, pp. 109–128, Mar. 2013. [Online]. Available: http://dx.doi.org/10.1007/s10817-013-9278-5
11. D. Déharbe, P. Fontaine, Y. Guyof, and L. Voisin, "Integrating SMT solvers in Rodin," *Science of Computer Programming*, vol. 94, Part 2, no. 0, pp. 130–143, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016764231400183X
12. C. Barrett and S. Berezin, "CVC Lite: A new implementation of the cooperating validity checker," in *16th CAV*. Springer, Jul. 2004, pp. 515–518, LNCS 3114.
13. J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin, "An open extensible tool environment for Event-B," in *8th Int'l. Conf. Formal Engineering Methods*. Springer, Nov. 2006, pp. 588–605, LNCS 4260. [Online]. Available: http://dx.doi.org/10.1007/11901433_32
14. C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard version 2.0," in *8th SMT Workshop*, 2010. [Online]. Available: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf
15. F. Immler, "Formally verified computation of enclosures of solutions of ordinary differential equations," in *Proc. 5th NASA Formal Methods Workshop*, 2014. [Online]. Available: http://home.in.tum.de/~immler/documents/immler2014enclosures.pdf
16. S. Harutunian, "Formal verification of computer controlled systems," Ph.D. dissertation, University of Texas, Austin, May 2007. [Online]. Available: https://www.lib.utexas.edu/etd/d/2007/harutunians68792/harutunians68792.pdf
17. R. Kurshan and K. McMillan, "Analysis of digital circuits through symbolic reduction," *IEEE Trans. CAD*, vol. 10, no. 11, pp. 1356–1371, Nov. 1991. [Online]. Available: dx.doi.org/10.1109/43.97615
18. L. Hedrich and E. Barke, "A formal approach to nonlinear analog circuit verification," in *ICCAD*, 1995, pp. 123–127. [Online]. Available: http://dl.acm.org/citation.cfm?id=224841.224870
19. M. R. Greenstreet, "Verifying safety properties of differential equations," in *8th CAV*, Jul. 1996, pp. 277–287. [Online]. Available: http://dx.doi.org/10.1007/3-540-61474-5_76

20. W. Hartong, L. Hedrich, and E. Barke, "Model checking algorithms for analog verification," in *39th DAC*, Jun. 2002, pp. 542–547. [Online]. Available: dx.doi.org/10.1109/DAC.2002.1012684

21. T. Dang, A. Donzé, and O. Maler, "Verification of analog and mixed-signal circuits using hybrid system techniques," in *5th FMCAD*, Nov. 2004, pp. 21–36. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30494-4_3

22. Z. J. Dong, M. H. Zaki, G. Al-Sammane, S. Tahar, and G. Bois, "Checking properties of PLL designs using run-time verification," in *Int'l. Conf. Microelectronics*, 2007, pp. 125–128. [Online]. Available: http://dx.doi.org/10.1109/ICM.2007.4497676

23. A. Jesser and L. Hedrich, "A symbolic approach for mixed-signal model checking," in *ASPDAC*, 2008, pp. 404–409. [Online]. Available: http://dl.acm.org/citation.cfm?id=1356802.1356903

24. M. Althoff, A. Rajhans *et al.*, "Formal verification of phase-locked loops using reachability analysis and continuization," *Comm. ACM*, vol. 56, no. 10, pp. 97–104, Oct. 2013. [Online]. Available: http://doi.acm.org/10.1145/2507771.2507783

25. H. Lin, P. Li, and C. J. Myers, "Verification of digitally-intensive analog circuits via kernel ridge regression and hybrid reachability analysis," in *50th DAC*, 2013, pp. 66:1–66:6. [Online]. Available: http://doi.acm.org/10.1145/2463209.2488814

26. H. Lin and P. Li, "Parallel hierarchical reachability analysis for analog verification," in *51st DAC*, 2014, pp. 150:1–150:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593178

27. G. Frehse, C. L. Guernic *et al.*, "SpaceEx: Scalable verification of hybrid systems," in *23rd CAV*, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_30

28. J. Wei, Y. Peng, G. Yu, and M. Greenstreet, "Verifying global convergence for a digital phase-locked loop," in *13th FMCAD*, Oct 2013, pp. 113–120. [Online]. Available: http://dx.doi.org/10.1109/FMCAD.2013.6679399

29. J. Marques-Silva and K. Sakallah, "GRASP: a search algorithm for propositional satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506–521, May 1999. [Online]. Available: http://dx.doi.org/10.1109/12.769433

30. R. Gamboa, "Mechanically verified real-valued algorithms in ACL2," Ph.D. dissertation, University of Texas at Austin, 1999.

31. J. Crossley, E. Naviasky, and E. Alon, "An energy-efficient ring-oscillator digital PLL," in *Custom Integrated Circuits Conf.*, Sep. 2010. [Online]. Available: http://dx.doi.org/10.1109/CICC.2010.5617417

Appendix

We found 3 small mistakes in our paper after the submission.

1. Page 2, paragraph 3.
   We've forgotten a citation for Harrison and Théry's "skeptical" approach. Here we give the citation:
   Harrison, J., and Théry, L. A skeptic's approach to combining HOL and Maple. Journal of Automated Reasoning 21 (1998), 279-294.
2. Page 7, paragraph 5.
   We have mistakenly understood Z3's way of handling divide by 0. Instead of considering the quotient to be an unspecified number, Z3 considers it to be any real number, which naturally include ACL2's interpretation of the quotient being 0. Therefore, our concern for the mismatch is not necessary. However, the "reciprocal" function we introduced is still sound.
3. Page 13, paragraph 3.
   [24]'s work does work for ranges of parameters.