

Integrating SMT with Theorem Proving

Verification of Analog and Mixed-Signal Circuits

Mark R. Greenstreet

October 22, 2019

- Motivating Examples
- Smtlink: integrating Z3 into ACL2
- Examples



Unless otherwise noted or cited, these slides are copyright 2019 by Mark Greenstreet and are made available under the terms of the Creative Commons Attribution 4.0 International license <http://creativecommons.org/licenses/by/4.0/>

Motivation

- Analog and Mixed Signal Design
 - ▶ “Linear Intent”
 - ▶ A digital phase-locked loop
 - ▶ The Rambus Oscillator
 - ▶ An asynchronous pipeline (timed circuit)
- Examples from our Machine-Learning Friends
 - ▶ Formalizing vector spaces: The Cauchy-Schwartz Lemma
 - ▶ Reasoning about convexity: Nesterov’s theorem

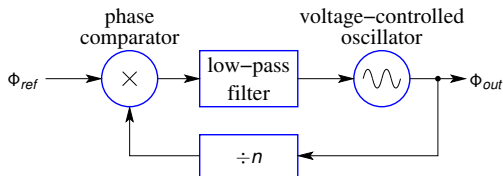


Linear Intent

- All big chips today include analog functions: clock multiplication and generation, high data-rate serial links, power management, support for sensors and actuators including video, RF, and audio.
- Kim *et al.* [KJLH09] recognized that
 - ▶ the circuits for analog and mixed-signal blocks are highly non-linear,
 - ▶ the intended functionality is nearly always linear.
 - ▶ there are very good, rigorous, mathematical methods for reasoning about linear systems.
- The global convergence problem:
 - ▶ Linear models apply in a small neighborhood of the intended operating region.
 - ▶ What happens when the circuit is not in this neighborhood:
 - ★ E.g. at start-up or following a mode switch?
 - ★ Is the circuit guaranteed to converge to the desired operating region?



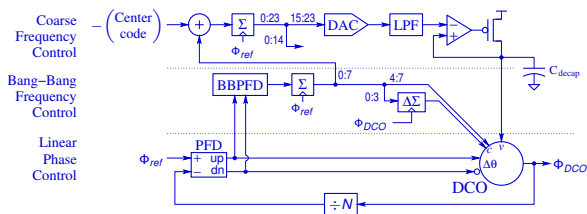
Linear Intent Example: a simple phase-locked loop



- Φ_{out} should be periodic with n times the frequency as Φ_{ref} and with a fixed phase relationship.
- Each component is highly non-linear at the SPICE-model level, but nearly linear in the domains of interest.
 - ▶ The voltage-controlled oscillator (VCO).
 - ★ Intent: a linear voltage-to-frequency converter.
 - ▶ The divider – a **digital** circuit:
 - ★ Intent: a scalar multiplication in the frequency domain by $1/n$.
 - ▶ The phase comparator: an analog multiplier – **how is that linear?!**
 - ★ Intent: in the frequency domain, it outputs the sum and difference of the input frequencies.
 - ▶ The low-pass filter – finally, something that's linear in the voltage domain.
 - ★ Intent: a linear integral-of-phase-to-voltage converter.
 - ★ Note: phase is the integral of frequency. Integration is linear.



A modern, “digital” phase-locked loop (PLL)

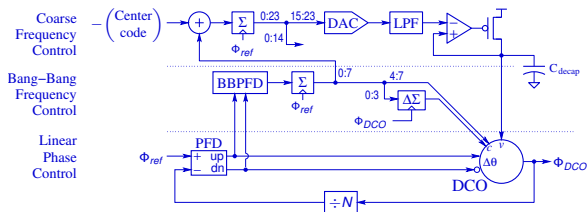


Mixed-signal design is the dominant paradigm.

- Digital circuits replace analog ones wherever possible.
- Low operating voltages, quirky transistors, and large device-to-device variation make analog design difficult in deep-submicron processes.
- Digital circuits are reliable, reproducible, and programmable.
- Digital circuits can be smaller than their analog counterparts
 - ▶ E.g. a 24-bit digital accumulator can be smaller than the capacitor needed for an analog integrator.



A motivational tale: global convergence of a PLL

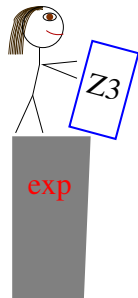


- The digital phase-locked loop (PLL) adjusts the frequency of the digitally-controlled oscillator (DCO) to match N times the reference, F_{ref} .
- **Global convergence:** will the digital PLL correctly converge to the desired frequency from any start up condition?
- Global convergence is important at initial start up, and when changing clock frequency, operating voltage, restarting a communication channel, etc.
- Design from [CNA10].



A motivational tale: SpaceEx and Z3

- Jijie Wei proved convergence of the digital PLL using SpaceEx [WPYG13].
 - ▶ Proof involves about 30 “reachability lemmas”.
 - ▶ Fixed design parameters.
- Yan Peng verified convergence using Z3:
 - ▶ Simplified model, but parameters remain symbolic.
 - ▶ Use Riccati equation to obtain a candidate Lyapunov function.
 - ▶ Use Z3 to verify that the proposed Lyapunov function establishes convergence.
 - ▶ It worked (yay!)
 - ▶ But it fails with a time-out for more realistic models.
- **It's easy to push an SMT solver off an exponential cliff.**



A motivational tale: Smtlink 1.0

- Manual proof for the fully parameterized digital PLL model:
 - ▶ It involves a recurrence (sum of a geometric series).
 - ▶ “Clever” rearrangement of the sum yields the key inequality.
 - ▶ A bunch of tedious algebra completes the proof.
- Smtlink 1.0 [PG15]:
 - ▶ Flatten non-recursive functions, add “hypotheses” to import facts from ACL2 logical world.
 - ▶ Use Z3 to discharge the flattened goal: $A \Rightarrow G_{Z3}$
 - ▶ Soundness:

$$\frac{A \vee G, \quad A \Rightarrow G_{Z3}, \quad (A \wedge G_{Z3}) \Rightarrow G}{G}$$

- ▶ It worked!
 - ★ So we tried bigger examples.
 - ★ Proving $(A \wedge G_{Z3}) \Rightarrow G$ (in ACL2) leads to time-outs.



Global Convergence and beyond

- **Exploiting linear intent requires establishing global convergence.**
 - ▶ Once global convergence has been shown, **linear-systems theory provides a mathematical formulation for reasoning about the design.**
 - ▶ Still need to watch out for **parametric failures**, e.g. failure to model interconnect resistance or coupling capacitance.
 - ▶ Still need to watch out for **residual non-linearities**
 - ▶ I believe that the formal methods community can contribute here as well: **We can deal with non-deterministic models.**
- **Global convergence is important beyond analog design:**
 - ▶ Control theory for **robotics** has its own challenges in reasoning about Lyapunov functions, i.e. continuous ranking functions convergence proofs.
 - ▶ **Machine learning uses numerical optimization** algorithms, often variations on gradient descent, where convergence is an issue.
- The focus of this tutorial:
 - ▶ **Using interactive theorem proving with SMT solvers for proving global convergence properties.**



Outline

○ Motivational Tale

➔ **Smtlink: integrating Z3 into ACL2**

- Architecture – flexible and extensible
- Soundness – a pragmatic approach
- Reflection & metaprogramming – make the common case easy
- Examples

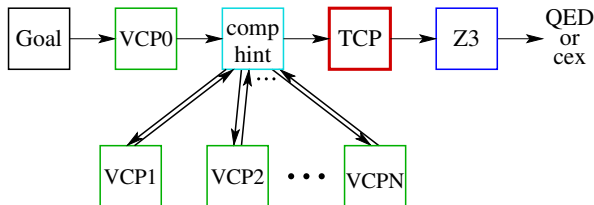


Smtlink 2.0

- Adds support for user defined data types: lists, alists, products,
- Uses ACL2's comprehensive support for reflection and metaprogramming to perform correct-by-construction transformations of the user's claim to a formula in the theories of the SMT solver.
 - ▶ Any auxiliary sub goals returned to ACL2 are “small”.
 - ▶ We determine when they are “small enough” by a combination of
 - ★ Simple “big- \mathcal{O} ” reasoning on the size of the term.
 - ★ Identify bottlenecks in real proofs.
- Ongoing improvements to the user interface.



Smtlink Architecture

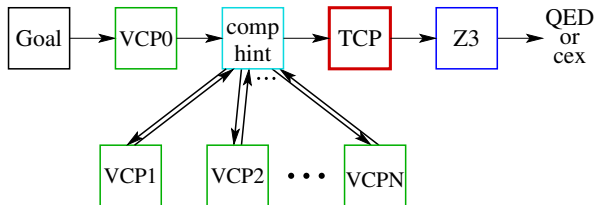


- VCP = “verified clause processor”
 - ▶ Replaces a clause with zero or more clauses.
 - ▶ Smtlink developer proves within ACL2:

```
(implies (my-eval replacement-clause)
(my-eval original-clause))
```
- TCP = “trusted clause processor”
 - ▶ **NO CORRECTNESS PROOF**
 - ▶ “Trust-tag” required – essentially adds `(sound TCP)` as an implicit hypothesis to any theorem with the trust tag.
- Comp. hint = “computed hint” examines current goal and applies hint.
 - ▶ Hints allow the user guide the ACL2 proof engine.
 - ▶ Does not affect soundness.



Smtlink Architecture



- VCP0: The user gives ACL2 the Smtlink “clause processor” hint.
 - ▶ VCP0 parses the user’s hint to provide a hint-object for subsequent steps of transforming the ACL2 goal into an SMT goal.
- The computed-hint \leftrightarrow VCP_I loop:
 - ▶ The goal is “tagged” with a hint that specifies what transformation step to apply next.
 - ▶ The computed hint reads the tag, and applies the ACL2 “hint” to invoke the requested clause processor.
 - ▶ The verified clause processor performs a correct-by construction transformation of the goal, and replaces the hint tag with a hint specifying the next step.
 - ★ These transformations may generate additional subgoals for ACL2.
 - ▶ The framework is flexible and extensible.
- The final step is to send the transformed goal to the SMT solver.

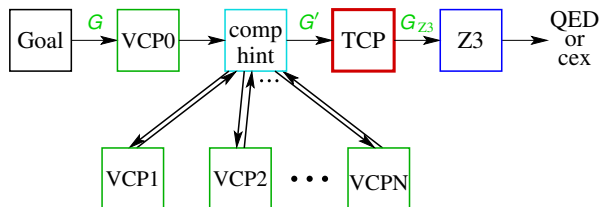


Outline

- Motivational Tale
- Smtlink: integrating Z3 into ACL2
 - Architecture – flexible and extensible
 - **Soundness – a pragmatic approach**
 - + Verified clause processors are sound.
 - + Correspondence of ACL2 and Z3 models.
 - + Types
 - + Reals vs. Rationals
 - + Functions
 - + Proof construction (Smtlink doesn't)
 - Reflection & metaprogramming – make the common case easy
- Examples



Smtlink Soundness



- Let G be the original goal, G' be the transformed goal that is set to TCP, and G_{Z3} be G' as transliterated into the Z3.py API.
- The translation of G to G' is sound, because
 - ▶ All transformations are performed by **verified** clause processors.
 - ▶ We trust ACL2.
- The transliteration from G' to G_{Z3} is a very small, simple, lisp function. We subject it to intense code review and inspection.
- Smtlink queries Z3 for a satisfying assignment for (not G_{Z3})
 - ▶ **Does the absence of a counter-example to G_{Z3} ensure that G' and thus G are theorems?**



Soundness Sketch

- Key Idea: If G' is not a theorem, then there is a model of $\neg G'$ in the logic of ACL2.
 - ▶ We show that there is a model of $\neg G_{Z3}$ in the logic of Z3, i.e. $\neg G_{Z3}$ is satisfiable.
 - ▶ Construct the model of $\neg G_{Z3}$ from the model for $\neg G'$ by induction on the structure of the formula of G' .
- Issues:
 - ▶ The logic of **ACL2 is untyped**, the logic of **Z3 is many-sorted**.
 - ▶ The **rational** numbers in ACL2 formulas are represented by **real** numbers in Z3.
 - ▶ The formula for G' can include recursive functions that can't be represented in Z3.



Soundness Example

```
(defthm bogus
  (implies (and (rationalp x) (< x 5))
    (< (* x x) 17)))
```

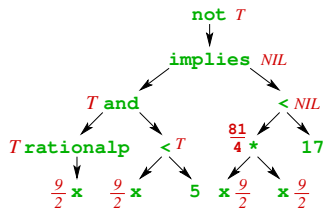
```
(not G'): (not (implies (and (rationalp x) (< x 5)) (< (* x x) 17)))
```



Soundness Example

```
(defthm bogus
  (implies (and (rationalp x) (< x 5))
    (< (* x x) 17)))
```

```
(not G'): (not (implies (and (rationalp x) (< x 5)) (< (* x x) 17)))
```

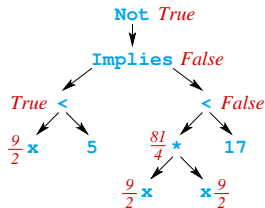
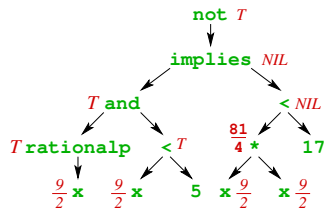


Soundness Example

```
(defthm bogus
  (implies (and (rationalp x) (< x 5))
    (< (* x x) 17)))
```

```
(not G'): (not (implies (and (rationalp x) (< x 5)) (< (* x x) 17)))
```

```
Not(Gz3): x = Real('x')
s = Solver()
s.add(Not(Implies(x < 5, x*x < 17)))
s.check() # returns sat
```



Now, we'll look at types, reals, and functions.



Soundness: Types

- While ACL2 is untyped, it does have *type-recognizer* functions, e.g. `integerp`, `consp`, ..., `my-user-type-p`.
- The TCP checks that every free-variable in G' has a corresponding type-hypothesis.
- If G' is not a theorem in ACL2, then $\neg G'$ has a model.
 - ▶ All of the hypotheses of G' must be satisfied in the model.
 - ★ Thus, this model assigns values of the “right” sort to each free variable in G' .
 - ▶ Inductively, all of the subterms of the model of $\neg G'$ must have the right types for Z3.
 - ★ Note: we assume that ACL2 functions satisfy type-theorems that correspond to the typing rules of Z3 for basic types.
 - ★ Example: the sum of two integers is an integer.
- While we trust TCP to make sure all terms are properly typed, this is in practice a “convenience” for the user. If TCP produces an improperly typed G_{Z3} , Z3 will fail with a user-hostile error message.



Soundness: Rationals vs. Reals

- Smtlink translates variables that satisfy `rationalp` to Z3 reals.
- Should be OK, because this just means that Z3 has models that ACL2 does not.

- ▶ e.g., in ACL2, we can prove

```
(thm (implies (rationalp x)
              (not (equal (* x x) 2))))
```

- ▶ Our translation into Z3 produces (roughly):

```
x = Real('x')
prove(Not(x*x == 2))
```

and the proof fails: $x = \sqrt{2}$ is a counter-example.

- Soundness is preserved
 - ▶ Smtlink may fail to prove a valid theorem. I.e. Smtlink is **incomplete**.
 - ▶ But it won't prove a non-theorem. I.e. Smtlink is **sound**.



Soundness: quantifiers

Note: Smtlink does not support ACL2's quantifier functions, e.g. `forall` and `exists`.

- We could try to prove:

```
(defun-sk square-p (x)
  (exists r (equal (* r r) x)))
(defthm 2-is-a-square
  (square-p 2)
  :hints(("Goal" :smtlink ...)))
```

But Smtlink will refuse to translate the goal because it doesn't have translations for `defun-sk` or `exists`.

- Could quantifiers be supported by Smtlink?
 - ▶ Future work – but only if we encounter motivating examples.



Soundness: Functions – how they are translated

- ACL2 has recursive functions.
 - ▶ Z3 only has uninterpreted functions.
- Non-recursive functions: Smtlink flattens calls.
- Recursive functions:
 - ▶ Expand to a user-specified depth.
 - ▶ Represent remaining calls with an uninterpreted functions.
 - ▶ Smtlink requires recursive functions to have a type signature
 - ★ function guard gives argument types.
 - ★ a “returns-theorem” gives the result type.
 - ★ the user can specify other constraints, justified by corresponding theorems they have already proven in ACL2.
 - ★ currently, Smtlink only supports polymorphism for a few built-in functions e.g. `cons`, `car`, `cdr`, `acons`, and `assoc`.
- Is this sound?



Soundness: Functions – why this is sound

- G' (the ACL2 term) can have recursive functions, and thus a model for $\neg G'$ can be countably infinite.
 - $\neg G_{Z3}$ (the Z3 term) corresponds to a finite “cap” of $\neg G'$.
 - ▶ Z3 can bind values for the uninterpreted function calls to match the values from a model for $\neg G'$.
 - ▶ Of, course, Z3 can find other models for $\neg G_{Z3}$ as well, that assign values for the results of uninterpreted functions that don't match the actual recursive function.
 - ▶ Again, Smtlink is sound, but not complete.
 - I'm an engineer; so, I'm OK with this justification.
 - Matt Kaufmann is a mathematician.
 - ▶ He's warned us about non-standard models, but believes it all works.
 - ▶ Yan Peng wants to graduate – I'm sure she'll come up with a proof.
- 😊



Soundness: What about Proof-Reconstruction?

- We see this as an orthogonal issue.
 - ▶ In principle, the final, trusted-clause processor could be replaced by a verified clause processor that uses proof reconstruction.
 - ▶ We would use the same (or similar) chain of verified clause processors to translate the user's goal, G , into a form that is amenable for the SMT solver, G' .
- Advantages of proof construction:
 - ▶ The moral argument: if we only trust a minimal theorem prover core, then we minimize the risk of unsoundness.
 - ▶ The engineering argument: trusted code is painful to write. Verified code requires less paranoia.
- Advantages of a little bit of trust
 - ▶ Adding new decision procedures is straightforward.
 - ▶ Avoids proof-reconstruction performance bottleneck.
 - ▶ The property that doesn't get verified is more likely to break a design than the one that was verified with a trusted SMT solver.
- Proof-reconstruction is “future-work” when there's a compelling use case:
 - ▶ E.g. Merijn Heule's proof for Boolean Pythagorean triples.



Outline

- Motivational Tale
- Smtlink: integrating Z3 into ACL2
 - Architecture – flexible and extensible
 - Soundness – a pragmatic approach
 - **Reflection & metaprogramming – make the common case easy**
 - + User-defined types
 - + Function expansion: lessons learned
- Examples



Types

- So far, we've looked at the “built-in” types: `booleanp`, `integerp`, `rationalp`, and `realp` (if you use ACL2(r)).
- Users can define their own types.
 - ▶ Smtlink 2.0 supports the ACL2 FTY package [SD15]: typed lists, typed alists, product types, tagged sums, etc.
 - ▶ Current work generalizes this to let the user define their own types without FTY
 - ★ We keep the “it just works” interface for FTY.
 - ★ But the in-progress version doesn't trust FTY.
- In progress:
 - ▶ Type-inference with a verified clause processor.
 - ▶ Independent of FTY – we support FTY, but we don't blindly trust it.
 - ▶ Discovering that many common lisp idioms can be translated to clearly typed equivalents automatically and efficiently.



Comparison with other integrations (always perilous)

- Isabelle/HOL Sledgehammer with SMT doesn't appear to provide such support (e.g. [BBP13]).
- CoqSMT seems to require the user to simplify their goal into something that directly corresponds to the SMT solver, i.e. the equivalent of our G' (based on the CoqSMT tutorial).
- We found that being able to work directly with goals stated using user-defined types is a **huge** productivity boost.
- Similar observations apply for function expansion.
- **ymmv**



Types Can Be Tricky

```
;; ACL2
(fty::deflist integer-list
  elt-type integer)
```

```
// Z3.py
IntegerList =
  Datatype('IntegerList')
IntegerList.declare('cons',
  ('car', IntSort()),
  ('cdr', IntegerList))
IntegerList.declare('nil')
IntegerList =
  IntegerList.create()
```

- In ACL2, `(car nil)` is `nil`,
- But in Z3, `IntegerList.car(IntegerList.nil)` is an unspecified integer.
 - ∴ A direct translation is unsound.
 - ▶ Smtlink 2.0 returns subgoals that the argument of each `car` is non-nil.
 - ▶ Discharging these subgoals can be a bottleneck, especially if the goal is huge after function expansion.



More Troubles with Types

- In ACL2, `cons`, `car`, and `cdr` are polymorphic, but in Z3, we need to use the specific version according to the type of the list.
 - ▶ In Smtlink 2.0, the user must wrap terms with fixing functions to let the TCP know which Z3 function to use.
 - ▶ This works, but it clutters function definitions and theorem statements.
 - ▶ Discharging these subgoals can be a bottleneck, especially if the goal is huge after function expansion.
- Smtlink solves these problems, and many more, by using clause processors with reflection and metaprogramming.



A bit about clause processors

Clause-processor provide a flexible mechanism for metaprogramming in ACL2.

- Clause processors operate on quoted terms.
- We can introduce an **evaluator** for these quoted terms (in a given environment).
- The clause processor returns a list of clauses.
- The evaluation a list of clauses is the conjunction of the evaluation of each clause.
- In any environment, the result of evaluating the list of clauses returned by a clause processor must imply the result of evaluating of the original clause.
 - ▶ For a **verified clause processor**, `vcp`, this implication is proven within ACL2.
 - ▶ Once verified, ACL2 can run the processor on a clause and use the result.
 - ▶ This provides a very fast and flexible mechanism for handling common proof patterns.



An example

A clause processor can use previously established facts in the ACL2 logical world. Consider:

;; What the user wrote

```
(define sum ((x integer-listp))
  :measure (len x)
  :returns (s integerp)
  (let ((xx (integer-list-fix x)))
    (if (endp xx) 0
        (+ (car xx) (sum (cdr xx))))))
```

How can we translate the body of `sum` to an SMT formula?



Typing Sum (work in progress, 1 of 4)

- Using reflection, a clause processor gets the meta-fact:

```
'(equal (sum x)
        (if (endp (integer-list-fix x)) 0
            (+ (car (integer-list-fix x))
                (sum (cdr (integer-list-fix x))))))
```

- ▶ This fact is returned from the ACL2 function `meta-extract` [KS17].
- ▶ For any such `fact`, `(my-eval fact)` is logically true, where `my-eval` is any evaluator.
- ▶ Thus, a verified clause processor can add `(not fact)` to the list of disjuncts of a clause without weakening the clause.
- ▶ Furthermore, the clause-processor can perform verified simplifications using these facts.



Typing Sum (work in progress, 2 of 4)

From

```
'(equal (sum x)
        (if (endp (integer-list-fix x)) 0
            (+ (car (integer-list-fix x))
                (sum (cdr (integer-list-fix x))))))
```

The clause processor obtains

```
'(if (integer-listp x)
    (equal (sum x)
          (if (endp (integer-list-fix x)) 0
              (+ (car (integer-list-fix x))
                  (sum (cdr (integer-list-fix x))))))
    t)
```

Because `fact` implies `(if p fact t)` for any `p`.



Typing Sum (work in progress, 3 of 4)

Constructing path conditions, and using meta-extract to instantiate `integer-list-fix-when-integer-listp` and similar theorems, the clause processor establishes:

```
'(if (integer-listp x)
      (equal (sum x)
             (if (equal x (integer-list-nil)) 0
                 (+ (integer-list-car x)
                    (sum (integer-list-cdr x))))))
t)
```

The `car` and `cdr` functions from the original function definition have been replaced with `integer-list-car` and `integer-list-cdr` which are translatable to operations on Z3 datatypes.



Typing Sum (work in progress, 4 of 4)

- The clause processor looks for an existing hypothesis to establish `(integer-listp x)`.
- With such a hypothesis, the definition of `sum` can be simplified to

```
'(equal (sum x)
        (if (equal x (integer-list-nil)) 0
            (int-int-+ (integer-list-car x)
                        (sum (integer-list-cdr x)))))
```

- This is fully type-specific, and ready for the transliteration to the Z3.py API.



User-Defined Types: Principles of Translation

- We found that representing ACL2 types in Z3 was a fruitless task.
 - ▶ The rampant polymorphism in Lisp precludes covering all the case.
- Representing Z3 types in ACL2 is straightforward.
 - ▶ And that's (usually) what the user intended anyway.
- Approach:
 - ▶ Transform the user's types into "close-enough" types that have Z3 equivalents.
 - ▶ Prove that the transformation is sound
 - ★ Mostly by clause processors that handle common lisp idioms – automatic and fast.
 - ★ As a last resort, return a subgoal to ACL2 to prove that the transformation was safe.



User-Defined Types: What we assume about Z3

- Algebraic data-types:

- ▶ If a destructor is applied to a term with a matching constructor, e.g.

```
IntegerList.car(IntegerList.cons(2, foo))
```

then the value from the constructor (e.g. 2) is returned.

- ▶ If a destructor is applied to any other term, e.g.

```
IntegerList.car(IntegerList.nil)
```

then an **arbitrary** value of the appropriate type is returned.

- Other type constructors (e.g. arrays) have similar rules.

- Smtlink checks that the type-specific constructor and destructor functions exist (in ACL2) and that they satisfy the properties given above.

- ▶ These requirements are discharged automatically if the names of the relevant theorems are given.
- ▶ This process is (in progress) automated when using the FTY package.



User-Defined Types: Summary

- ACL2 is untyped, and users employ a rich set of custom type-recognizers.
- Verified clause processors can derive types for terms and replace untyped functions with typed ones.
 - ▶ If the type of a term can't be determined, and error is reported.
 - ▶ If a side condition can't be established (e.g. `(consp x)`), then a subgoal is returned to ACL2.
 - ▶ This is work-in-progress, but so far, everything is very fast, and Smtlink supports a wide-range of the most common lisp idioms.
- Smtlink supports list, alist, product, tagged-sum, symbol, array, and uninterpreted types.
 - ▶ Smtlink 2.0 trusts FTY. FTY types are effectively built-in to the trusted core of Smtlink 2.0.
 - ▶ Smtlink-in-progress will provide macros to “import” FTY types into Smtlink, including all soundness checks. No trust assumptions about FTY are needed.



Functions: Overview

- Transformation of terms implemented with verified clause processors using `meta-extract`. Very much as with types.
- Recursive functions are translated to uninterpreted functions.
 - ▶ **Don't** expand in place!
 - ▶ Instead, add constraints of the form

```
(equal function-call  
function-body-instantiated-with-actuals)
```

for calls to a user specified depth.
 - ▶ This avoids exponential blow-ups in the size of the formula.
 - ▶ It also makes it **much** easier to discharge goals under an induction proof.
- In progress: type check each function when defined, and prove “SMT-typing” theorem.
 - ▶ This allows type inference to be done on smallish pieces of code.
 - ▶ Fast. More predicatable for the user. Better error messages.



Induction Example (1 of 2)

Let

```
(define sum-of-nats ((n natp))
  :returns (s natp)
  (if (zp n) 0
      (+ n (sum-of-nats (1- n)))))

(defthm closed-form-for-sum-of-nats
  (implies (natp n)
    (equal (sum-of-nats n)
           (/ (* n (1+ n)) 2))))
```

- `closed-form-for-sum-of-nats` is easily proven in ACL2 without Smtlink, but let's look at using Smtlink for the induction step anyway.



Induction Example (2 of 2)

The “interesting” induction-step goal:

```
(implies (and (integerp n)
              (<= 0 n)
              (equal (sum-of-nats (+ -1 n))
                    (+ (* 1/2 (+ -1 n))
                      (* (+ -1 n) 1/2 (+ -1 n))))))
         (equal (sum-of-nats n)
               (+ (* 1/2 n) (* n 1/2 n))))
```

- If the clause processor expanded for **both** occurrences of `sum-of-nats` in place, we'd just get a similar goal, relating `(sum-of-nats (- n 1))` and `(sum-of-nats (- n 2))`. This isn't very helpful.
- OTOH, by making `sum-of-nats` uninterpreted (in Z3) and adding constraints equating `(sum-of-nats n)` and `(sum-of-nats (- n 1))` to their respective instantiations of the function body, Z3 easily discharges the claim.
- **Conclusion:** Describing function instantiation as constraints on uninterpreted functions is “better” than performing in-place function expansion because it lets the SMT solver “see” more of the problem.



Outline

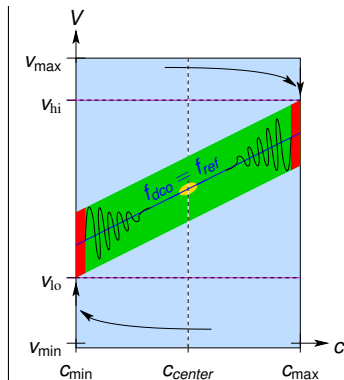
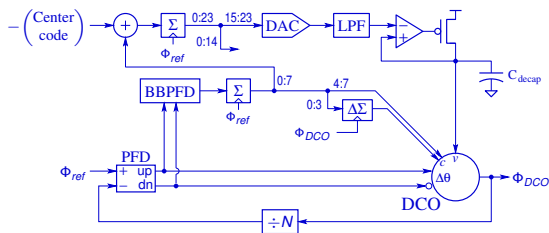
- Motivational Tale
- Smtlink: integrating Z3 into ACL2

➤ Examples

- Smtlink 1.0: global convergence for digital phase-locked loop
- Smtlink 2.0: verification of timed, asynchronous pipeline.
- Smtlink-in-progress: Cauchy-Schwartz lemma



The Digital PLL: will it converge?



- Model by non-linear recurrence: updates circuit state at rising edge of Φ_{ref} .

- Non-linearity from $f_{dco} = \frac{1 + \alpha V}{1 + \beta C} f_0$.



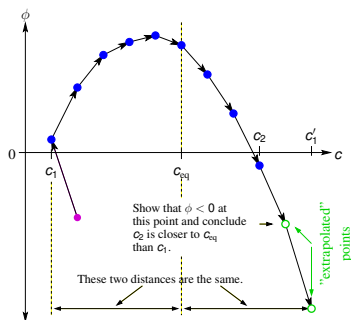
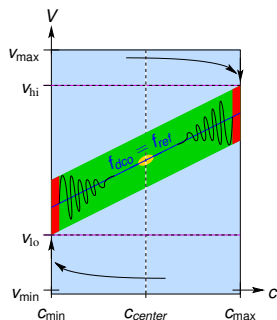
The Digital PLL: the recurrence

$$\begin{aligned}c(i+1) &= \min(\max(c(i) + g_c \operatorname{sgn}(\phi), c_{\min}), c_{\max}) \\v(i+1) &= \min(\max(v(i) + g_v(c_{\text{center}} - c(i)), v_{\min}), v_{\max}) \\\phi(i+1) &= \operatorname{wrap}(\phi(i) + (f_{\text{dco}}(c(i), v(i)) - f_{\text{ref}}) - g_\phi \phi(i)) \\f_{\text{dco}}(c, v) &= \frac{1 + \alpha v}{1 + \beta c} f_0 \\\operatorname{wrap}(\phi) &= \operatorname{wrap}(\phi + 1), \quad \text{if } \phi \leq -1 \\&= \phi, \quad \text{if } -1 < \phi < 1 \\&= \operatorname{wrap}(\phi - 1), \quad \text{if } 1 \leq \phi\end{aligned}$$

- The Digital PLL has a mode switch based on the sign of ϕ .
- Within a mode, we can solve for $c(i)$ and $\phi(i)$.



The Digital PLL: proof strategy



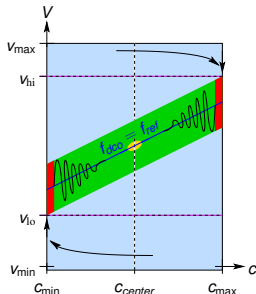
- Key observation (in the green, diagonal band):
Successive crossings of the c axis converge towards $c = 0$ until the crossing has $|c| \leq 3$.
- Proof (sketch):
 - ▶ The Digital PLL has a mode switch based on the sign of ϕ .
 - ▶ Solve for $\phi(i)$ for c one step closer to c_{eq} than the previous change, and show that ϕ has already changed sign.



The Digital PLL: the key inequality

Let

- $c_0(v)$ satisfies $f_{dco}(v, c_0(v)) = Nf_{ref}$.
- Δ_c = size of a step in the c control of the VCO.
- K_t = time-gain of linear-phase path. $0 < K_t < 1$.
- $\mu = \frac{f_0}{Nf_{ref}}$



The key inequality shows that (given a bunch of constraints on α , β , Δ_c , K_t , and μ) that for any $h \geq 3$:

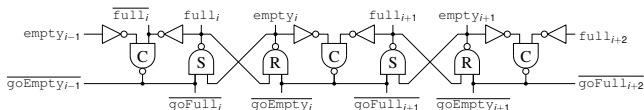
$$\begin{aligned} & (1 - K_t)^{-h} f_{dco}(v, c_0 - (1 - h)\Delta_c) \\ & + (1 - K_t)^{h-2} f_{dco}(v, c_0 - (h - 1)\Delta_c) \\ < & -\frac{9}{8}(1 - K_t)^{-3-h} \frac{\beta\Delta_c}{\mu(1+\alpha v)} \end{aligned}$$

Proving the inequality, using Smtlink 1.0, takes about 1 minute.

- The full proof has 75 lemmas, 10 discharged by Smtlink 1.0.
- We expect a simpler proof in the current Smtlink.



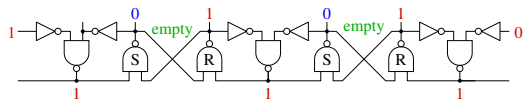
Verifying Timed Asynchronous Circuits



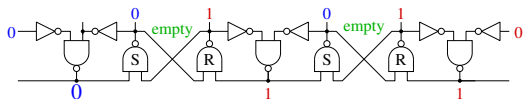
- Asp* pipelines [MJ⁺97] are simple, fast, and can be synthesized using the cells in standard libraries.
- They also have timing dependencies:
 - ▶ When $stage[i+1]$ is empty (i.e. not full) and $stage[i]$ is full (i.e. not empty), they are enabled to concurrently change.
 - ▶ Either one changing disables the action for both.
 - ▶ Must show that the difference in the transition times for the RS latches is less than the delay of the NAND gate (labeled C).
 - ▶ A few other constraints as well.
 - ▶ This is a simple and useful example to illustrate asynchronous circuits and timing dependencies.



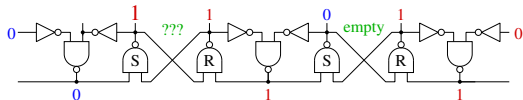
Verifying Timed Asynchronous Circuits: Example



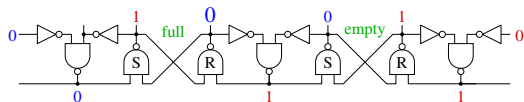
Verifying Timed Asynchronous Circuits: Example



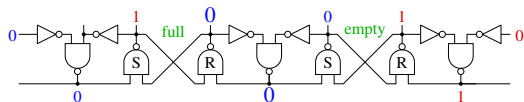
Verifying Timed Asynchronous Circuits: Example



Verifying Timed Asynchronous Circuits: Example



Verifying Timed Asynchronous Circuits: Example

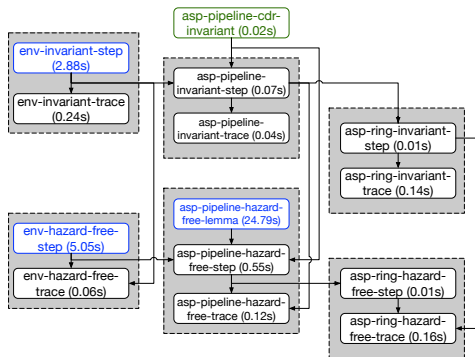


Modeling Timed Asynchronous Circuits

- We use a trace-based approach. [Dil88]
- A trace is a sequence of states.
- Each state is a mapping from signal-identifiers to signal values.
 - ▶ A signal-identifier is a hierarchical path name.
- Components are **trace recognizers**
 - ▶ A component must accept any behaviour of its inputs.
 - ▶ A component recognizes a trace if it allows the behaviour of its outputs.
- Timing constraints are modeled by including time in the state space. [AL92].



The correctness proof



One theorem for each key inductive result.

- Many of these theorems have one support lemma for the induction step.
- These support lemmas are proven with Smtlink with no further assistance.

Smtlink handles nearly all of the details – proofs are fun!



The Cauchy-Schwartz Lemma

$$\langle u, v \rangle^2 \leq \langle u, u \rangle \langle v, v \rangle$$

With equality iff u and v are co-linear

- If the range of the inner-product provides square-root, then

$$|\langle u, v \rangle| \leq \|u\| \|v\|$$

- A widely used theorem from linear algebra.
- We needed it for reasoning about convex functions for machine learning.
- Proof (by Carl Kwan) without Smtlink.



Cauchy-Schwartz: a key lemma

- A key lemma:

```
(defthm csl-when-v-not-zero
  (implies (vector-compatible u v)
    (let ((uu (inner-prod u u))
          (uv (inner-prod u v))
          (vv (inner-prod v v)))
      (<= (* uv uv) (* uu vv))))
  :hints(("Goal" ...)))
```

- Proof without Smtlink has nine supporting lemmas to walk the theorem prover through the derivation



Cauchy-Schwartz meets Smtlink

- The proof with Smtlink consists two theorems
 - ▶ The top-level theorem handles the `(vector-compatible)` hypothesis. ACL2 proves this theorem without Smtlink.
 - ▶ The main lemma is proven using Smtlink.
 - ★ Eight user-added “hypotheses” to give Z3 the facts it needs about vector operations.
- This is in-progress. We’re making the interface easier to use
 - ▶ Automatically extract type-signatures for functions that are defined using `define`.
 - ▶ Simplify “hypothesize” hints to support just stating the theorem name and the bindings for free variables.



What's a “hypothesize” hint

- Recall that it's easy to push an SMT solver over an exponential cliff.
 - ▶ It's tempting to tell the SMT solver everything you know and see if it will solve the problem.
 - ▶ ACL2 knows a lot about the functions you've defined – telling the SMT solver everything is a bad idea.
- OTOH, some of existing theorems may be required to prove the current one.
- Smtlink provides a “hypothesize” hint to convey facts from the ACL2 logical world to the SMT solver:

```
:hypothesize (  
  (fact1) ;; fact1 added as hypothesis for Z3  
           ;; and returned as subgoal to ACL2  
  (fact2  
    :hints (:use (:instance some-theorem  
                  (x (+ (* 2 a) b))  
                  (y (- a (* 2 b))))))))
```



- Some simple examples from the on-line tutorial:

http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=SMT_TUTORIAL

- ▶ Polynomial inequalities.
 - ▶ User-defined types: products, lists, and lists
- The Cauchy-Schwartz Inequality



Summary

- Smtlink integrates the Z3 SMT solver into ACL2.
 - ▶ Supporting other solvers such as Yices or CVC4 should be straightforward, esp. support for SMT-LIB.
 - ▶ Currently, using the Python bindings of Z3.py provides a flexible interface for prototyping.
- Reflection and metaprogramming are powerful mechanisms for transforming the goal the user wants to prove into a formula that the SMT solver can discharge.
- Support for user-defined data types means SMT is for more than just numbers.



Future Work

Short to Medium Term

- Complete the current Smtlink.
 - ▶ Add counter-example generation with user-defined types.
 - ▶ Add symbolic simulation.
 - ▶ When we encounter a tedious proof, we almost always respond:
There's a metaprogramming solution for that!.
- Circuits and Robots
 - ▶ Automatic differentiation for fixpoint algorithms.
 - ▶ Integrate interval verification algorithms into SMT framework.
 - ▶ Analyse and verify interesting designs.



Thanks

- This work has benefitted from my wonderful graduate students and many others including: Itrat Akhter, Chris Chen, Leo de Moura, Ian Jones, Matt Kaufmann, Ian Mitchell, Yan Peng, **Your name goes here**, Justin Reiher, Mark Schmidt, Margo Seltzer, Jijie Wei, Chao Yan, and Suwen Yang.
- This work would not have been possible without the financial support of NSERC, Intel, Oracle, and ICICS UBC.

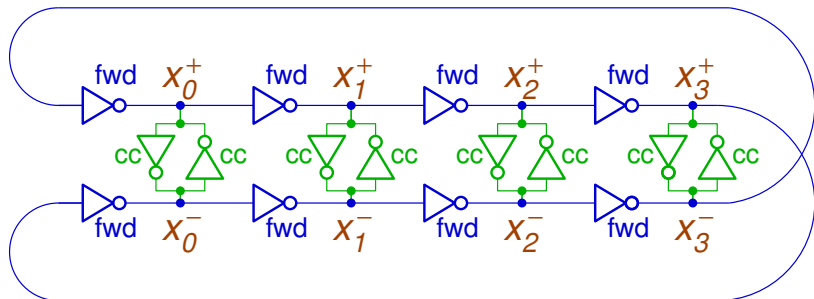


Outline

- Smtlink: integrating Z3 [MB08] into ACL2
- Current and recent hardware verification research
 - Interval Verification Algorithms
- Future work and wild ideas



The Rambus Ring-Oscillator (RRO)



Generates multiple, evenly spaced, differential phases.

Easily modified to produce a VCO.

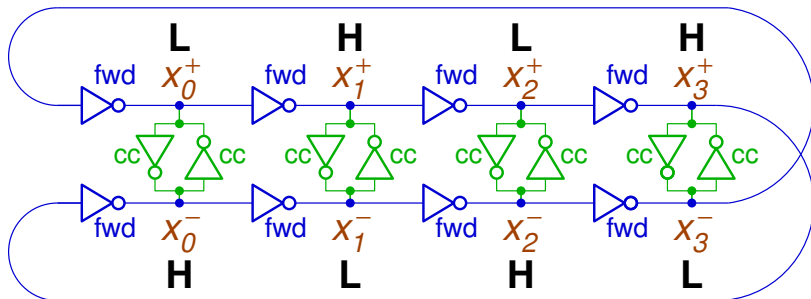
Can operate at high-frequencies:

2-stage version has 4 inverter-delay period.

Will it start-up reliably?



The Rambus Ring-Oscillator (RRO)



$t = 0:$

X_0^+ —

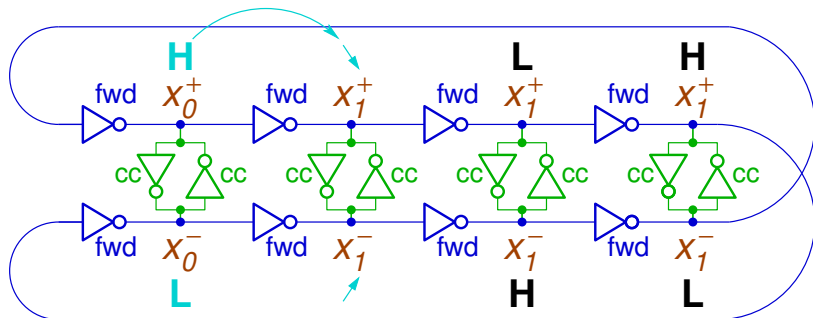
X_1^+ —

X_2^+ —

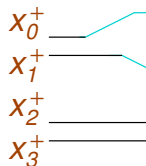
X_3^+ —



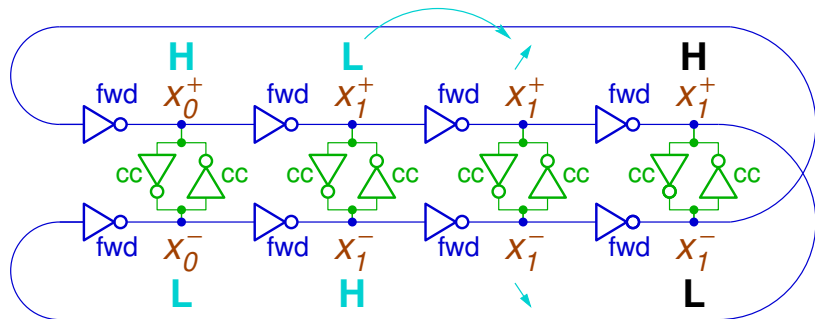
The Rambus Ring-Oscillator (RRO)



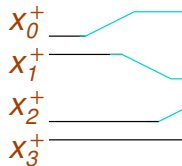
$t = 1:$



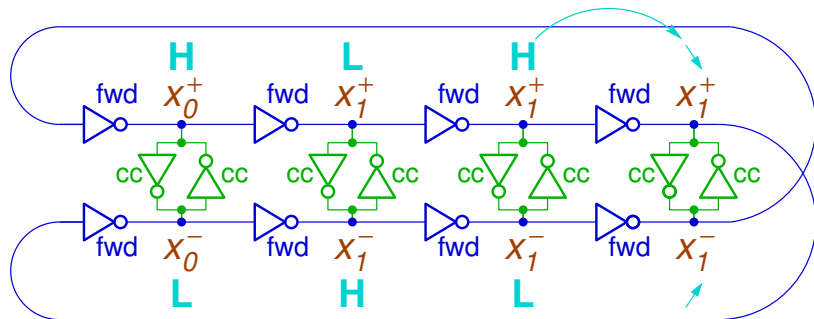
The Rambus Ring-Oscillator (RRO)



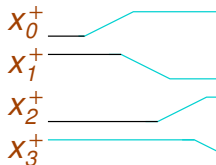
$t = 2:$



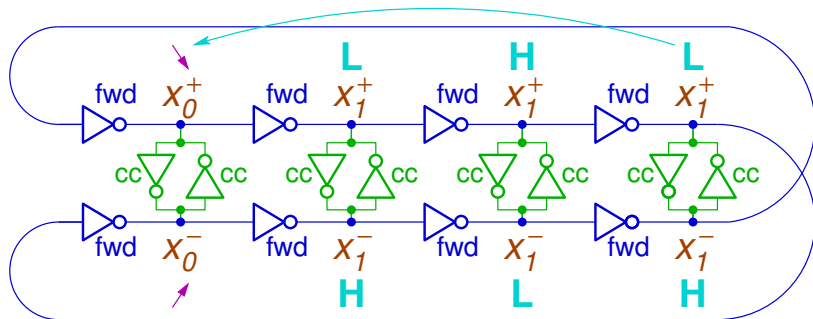
The Rambus Ring-Oscillator (RRO)



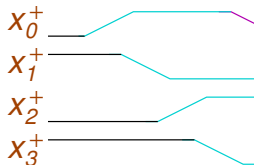
$t = 3:$



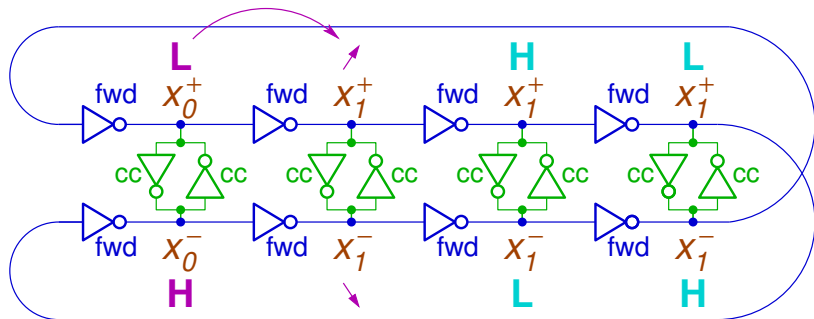
The Rambus Ring-Oscillator (RRO)



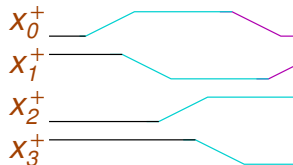
$t = 4:$



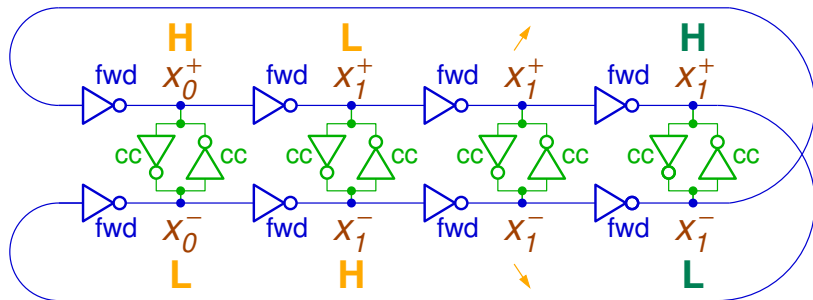
The Rambus Ring-Oscillator (RRO)



$t = 5:$



The Rambus Ring-Oscillator (RRO)



$t = \dots$



The Verification Challenge

- Proposed by Jones et al. [JKK08].
- Once the oscillator is running
 - ▶ It is easy to show that the oscillation mode is stable (dynamical systems theory, eigenvalue analysis).
- Will the oscillator start from all initial conditions?
 - ▶ Failures have been observed for real chips in spite of extensive SPICE simulation before fabrication.
 - ▶ Known to depend on the transistor sizes in the inverters.
 - ▶ **What transistor sizes guarantee proper start-up?**



Oscillator Start-up: Solution

- Find all DC equilibria points and check each for stability.
 - ▶ Pencil-paper-and-Matlab solution in [GY08].
 - ▶ First fully automated solution in [ARG19].
- Approach: Krawczyk's operator, an interval arithmetic verification algorithm (see [Rum99]).
 - ▶ Use Newton's method to estimate solution starting from center of hyper-rectangle.
 - ▶ Use inverse of Jacobian as pre-conditioner matrix (not interval), and then compute interval bounds for next.
 - ▶ Relationship of current and next hyper-rectangles determines the existence of a unique solution or no solution in the hyper-rectangle, or provides a smaller hyper-rectangle to search.
- Results: solved ring-oscillator problem with a state-of-the-art transistor model (MVS [RA15]).
 - ▶ dReal and Z3 time out on all test cases.



Outline

- Smtlink: integrating Z3 [MB08] into ACL2
- Current and recent hardware verification research
- ➡ Future work and wild ideas



Wild and crazy ideas

- Machine learning (with Mark Schmidt)
 - ▶ Convergence arguments for the optimization algorithms used in learning are **very** similar to those for AMS circuits.
 - ★ Can we mechanize these proofs?
 - ★ With confidence in the proofs, can we make more aggressive algorithms?
 - ★ Can we make the proof process as “easy” as writing up the informal arguments in latex?
 - ▶ “Tending α to 1...”
 - ★ A remark by Nesterov in a proof about convex functions that Carl Kwan proved using ACL2.
 - ★ Can we use remarks like this to obtain useful priors for a machine-learning based proof search?
- Operating systems (with Margo Seltzer)
 - ▶ Can we model OS functions in a language like Unity (guarded commands)?
 - ▶ Can we prove the OS correct?
 - ▶ Can we synthesize efficient code?
 - ▶ Can we we design an OS that is distributed, parallel, heterogeneous, and secure from the start?



-  Martín Abadi and Leslie Lamport, *An old-fashioned recipe for real time*, Proceedings of the REX Workshop, “Real-Time: Theory in Practice” (J.W. de Bakker et al., eds.), Springer, 1992, LNCS 600.
-  Itrat A. Akhter, Justin Reiher, and Mark Greenstreet, *Finding all dc operating points using interval arithmetic based verification algorithms*, Proceedings of the 25th Design, Automation and Test, Europe Conference, March 2019.
-  Jasmin C. Blanchette, Sascha Bohme, and Lawrence C. Paulson, *Extending Sledgehammer with SMT solvers*, Journal of Automated Reasoning **51** (2013), no. 1, 109–128.
-  John Crossley, Eric Naviasky, and Elad Alon, *An energy-efficient ring-oscillator digital PLL*, Proceedings of the Custom Integrated Circuits Conference (CICC’2010), September 2010.
-  David L. Dill, *Trace theory for automatic hierarchical verification of speed-independent circuits*, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, 1988, Published in book form



as part of the ACM Doctoral Dissertation Award Series by the MIT Press, Cambridge, MA, 1989.



Mark R. Greenstreet and Suwen Yang, *Verifying start-up conditions for a ring oscillator*, Proceedings of the 18th Great Lakes Symposium on VLSI (GLSVLSI'08), May 2008, pp. 201–206.



Kevin D. Jones, Jaeha Kim, and Victor Konrad, *Some “real world” problems in the analog and mixed-signal domains*, Proc. Workshop on Designing Correct Circuits, April 2008.



Jaeha Kim, Metha Jeeradit, Byongchan Lim, and Mark A. Horowitz, *Leveraging designer's intent: a path toward simpler analog CAD tools*, Proceedings of the Custom Integrated Circuits Conference (CICC'2009), September 2009, pp. 613–620.



Matt Kaufmann and Sol Swords, *Meta-extract: Using existing facts in meta-reasoning*, Proceedings 14th International Workshop on the ACL2 Theorem Prover and its Applications, May 2017, pp. 47–60.





Leonardo Moura and Nikolaj Bjørner, *Z3: An efficient SMT solver*, Tools and Algorithms for the Construction and Analysis of Systems (C.R. Ramakrishnan and Jakob Rehof, eds.), Lecture Notes in Computer Science, vol. 4963, Springer Berlin Heidelberg, 2008, pp. 337–340.



Charles E. Molnar, Ian W. Jones, et al., *A FIFO ring oscillator performance experiment*, Proceedings of the Third International Symposium on Advanced Research in Asynchronous Circuits and Systems, IEEE Computer Society Press, April 1997, pp. 279–289.







Yan Peng and Mark Greenstreet, *Integrating SMT with theorem proving for analog/mixed-signal circuit verification*, NASA Formal Methods (Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, eds.), Springer International Publishing, 2015, pp. 310–326.



Y. Peng, I. W. Jones, and M. R. Greenstreet, *Finding glitches using formal methods*, Proceedings of the 22nd International Symposium on Asynchronous Circuits and Systems, IEEE Computer Society, May 2016, pp. 45–46.



-  Shaloo Rakheja and Dimitri Antoniadis, *MVS nanotransistor model (silicon) 1.1.1*, nanohub.org, December 2015.
-  S.M. Rump, *INTLAB - INTerval LABoratory*, Developments in Reliable Computing (Tibor Csendes, ed.), Kluwer Academic Publishers, Dordrecht, 1999, <http://www.ti3.tu-harburg.de/rump/>, pp. 77–104.
-  Sol Swords and Jared Davis, *Fix your types*, Proceedings 13th International Workshop on the ACL2 Theorem Prover and its Applications, October 2015, pp. 3–16.
-  Jijie Wei, Yan Peng, Ge Yu, and Mark Greenstreet, *Verifying global convergence for a digital phase-locked loop*, Proceedings of the 13th Conference on Formal Methods in Computer Aided Design (FMCAD'2013), Oct 2013, pp. 113–120.

