Hardware Verification Using Theorem Proving and SMT/SAT Solving

Yan Peng[1]

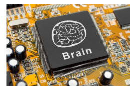
[1]Department of Computer Science
University of British Columbia

November 7th 2018

1. Motivation
2. Smtlink
3. AMS Verification
4. Asynchronous Circuit Verification
5. Glitch Hunting
6. The Exciting Future Work

The Quest of FV of Timed/Continuous Systems

1. Formal verification for digital circuits involving only discrete states has been extensively researched
2. Formal verification for timed and continuous systems is less mature than formal verification of digital hardware



   1. Huge area of applications: chip designs, robotics, autonomous cars, neuromorphic chip designs and other cyber-physical systems
   2. Reachability analysis
      1. over-approximation can be too large, refining over-approximation might lead to run-time and memory issues
      2. Usually the analysis start with a fixed set of parameters

Bridging Theorem Proving and SMT/SAT Solving

1. We revisit the rigorous approach of using theorem proving and aim to analytically verify timed and continuous systems
    1. Can prove a system for a range of parameters
    2. Doesn't have the over-approximation problem
    3. "Our designers are not going to learn these theorem provers ..."
    4. Due to good reason ... theorem proving requires excessive manual work, and ... expertise
2. We address this problem by combining theorem proving with SMT/SAT solving
    1. Theorem provers serve as a problem and property modeling system and induction proof engine
    2. SMT/SAT solvers crack out the details in large flattened lemma instances

1. Motivation
2. Smtlink
3. AMS Verification
4. Asynchronous Circuit Verification
5. Glitch Hunting
6. The Exciting Future Work

The ACL2 Theorem Prover

1. **A** **C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp
2. The ACL2 theorem prover is both interactive and automatic
3. ACL2 uses a subset of Common Lisp, which allows serious programming
4. The use of `clause-processors`, `computed hints`, `meta-extract` to allow logically sound extensions of the theorem prover
5. ACL2 emphasize large system verification and has various language supports for improving performance

Smtlink

ACL2::projects

Smtlink

[books]/projects/smtlink/doc.lisp

SMT Package

Tutorial and documentation for the ACL2 book, Smtlink.

Introduction

A framework for integrating external SMT solvers into ACL2 based on the ACL2::clause-processor and the `ACL2::computed-hints` mechanism.

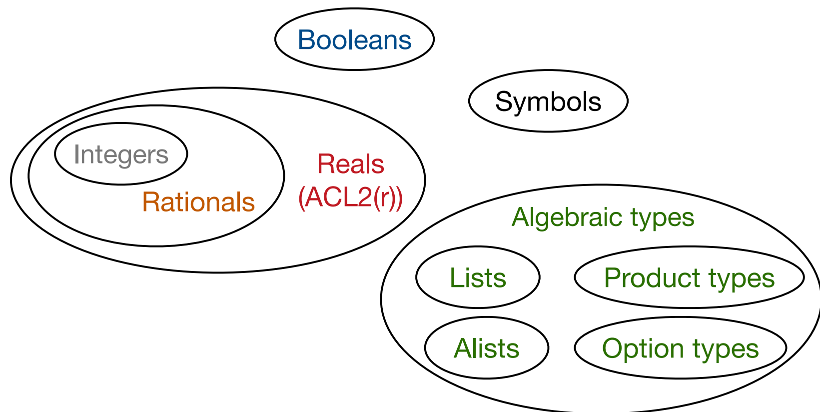
Overview

`Smtlink` is a framework for representing suitable ACL2 theorems as a SMT (Satisfiability Modulo Theories) formula, and calling SMT solvers from within ACL2.

A sound framework for integrating SMT solvers into the ACL2 theorem prover.[1]

[1] Y. Peng and M. R. Greenstreet. "Smtlink 2.0". In: *15th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2018)*. 2018.

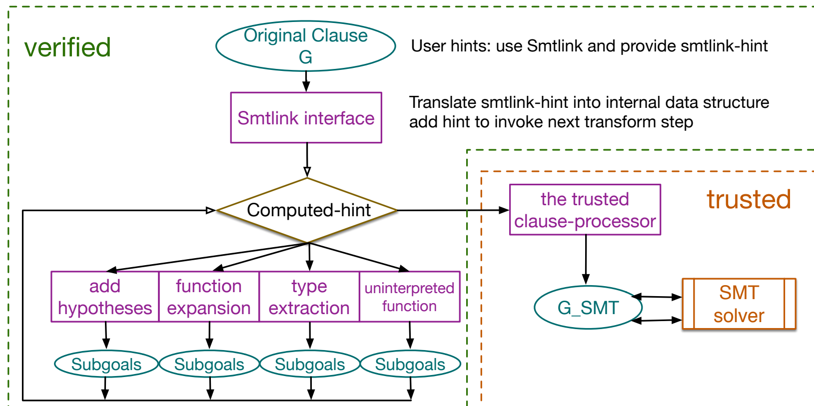
What's Supported in Smtlink



1. These cover a moderate amount of datatypes that are required for modeling systems

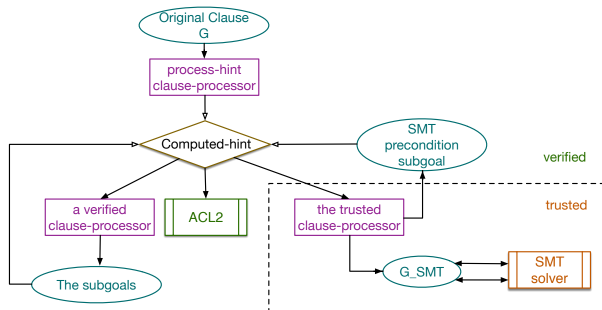
The Architecture

The architecture is both extensible and has a compelling argument for soundness



Verified clause-processors transform
ACL2 goal into SMT theories.
Each verified clause-processors adds a hint
indicating which step to take next.

The Architecture - Cont'd

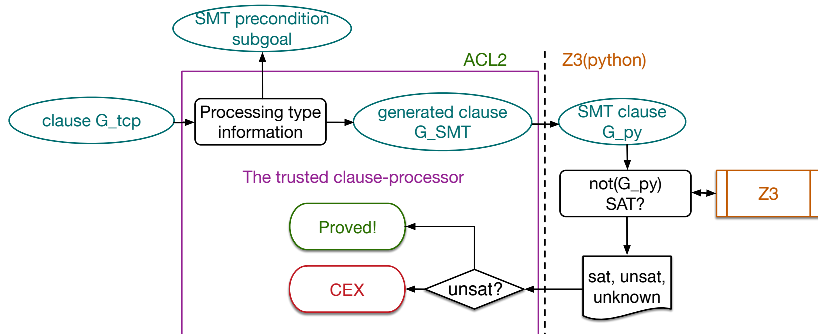


smt-architecture table

step tag	next clause-processor
process-hint	add-hypo-cp
add-hypo	expand-cp
expand	type-extract-cp
type-extract	uninterpreted-fn-cp
uninterpreted	smt-trusted-cp
uninterpreted-custom	smt-trusted-cp-custom

1. Each step is a verified clause-processor that can be configured through a single table
2. Only the last step uses a trusted clause-processor

The Trusted Clause Processor



1. What's not verified? The trusted clause-processor, Z3py interface class, and Z3
2. SMT precondition subgoals: subgoals that have to be satisfied to ensure soundness.

Counter-example Generation

types	counter-example examples
booleans	`((X NIL))`
integers	`((X 0))`
rationals	`((X 1/4))`
algebraic numbers	`((Y (CEX-ROOT-OBJ Y STATE (+ (^ X 2) (- 2)) 1)) (X -2))`
symbols	`((X (SYM 0)))`
lists	`((L (CONS 0 (CONS 0 NIL))))`
alists	`((L (K SYMBOL (SOME 0))))`
product types	`((S2 (SANDWICH 0 (SYM 2))) (S1 (SANDWICH 0 (SYM 1))))`
option types	`((M2 (SOME 0)) (M1 (SOME 0)))`

1. Algebraic numbers are represented by the kth root of some polynomial
2. The `(K s v)` for alists represents an array mapping any values of s sort/type into a constant value (or an expression) v.
3. Currently evaluable counter-examples are booleans, integers and rationals

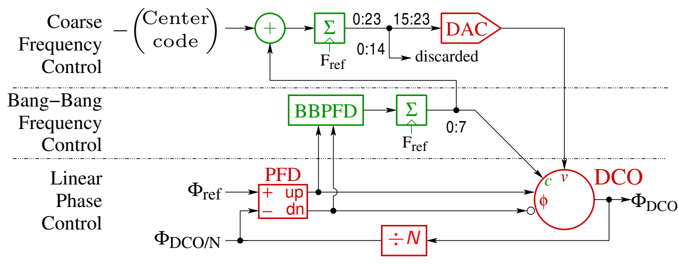
Summary

In summary,

1. I built a novel sound framework for integrating the Z3 SMT solver into the ACL2 theorem prover
2. There are several highlights of `Smtlink`:
   1. This framework itself is mostly verified, leading to a compelling argument of soundness
   2. It supports a substantial number of datatypes and SMT theories, therefore can find use in a large number of applications
   3. Counter-examples are returned back into the ACL2 theorem prover for further scrutiny
   4. Coming together with ACL2 available at: https://github.com/acl2/acl2/tree/master/books/projects/smtlink with documentation at: http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=SMT____SMTLINK

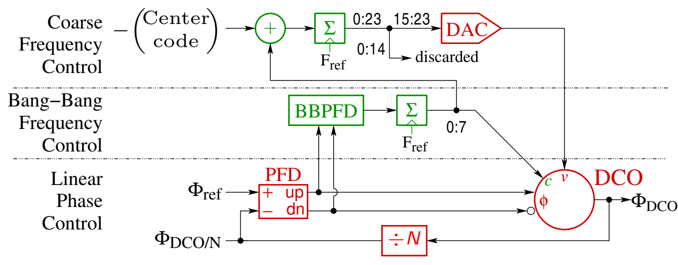
1. Motivation
2. Smtlink
3. AMS Verification
4. Asynchronous Circuit Verification
5. Glitch Hunting
6. The Exciting Future Work

The digital Phase-Locked Loop example[CNA10]



- A PLL is a feedback control system that, given an input reference clock f_{ref}, it outputs a clock at a frequency f_{DCO} that's N times of the input clock frequency and aligned with the reference in phase.
- Analog/Mixed-Signal design are composed of both analog and digital circuits.

The digital Phase-Locked Loop example[CNA10]



- We published an early version of this proof[2] using two approaches – hybrid automaton and Lyapunov functions:
  1. Main limitation with hybrid automaton: Reasoning about a fixed design
  2. Main limitation with Lyapunov approach: model is simplified

[2] J. Wei et al. "Verifying global convergence for a digital phase-locked loop". In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 113–120. DOI: 10.1109/FMCAD.2013.6679399.

Modelling the digital PLL

- The digital PLL is naturally modelled using non-linear recurrences that update the state variables on each rising edge of ϕ_{ref}.

$$
\begin{aligned}
c(i+1) &= next_c(c(i), v(i), \phi(i)) \\
v(i+1) &= next_v(c(i), v(i), \phi(i)) \\
\phi(i+1) &= next_\phi(c(i), v(i), \phi(i))
\end{aligned}
$$
[3]

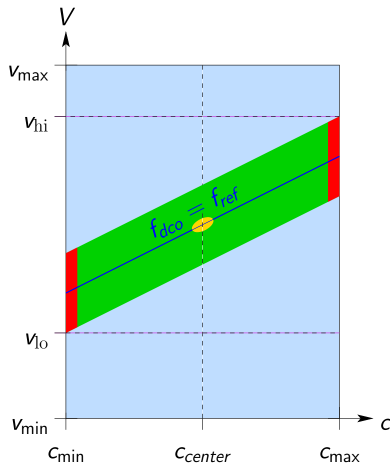
[3] Three state variables: capacitance setting c (digital), supply voltage v (linear), phase correction ϕ (time-difference of digital transitions).

Modelling the digital PLL

- In more details,

$$
\begin{aligned}
c(i+1) &= \mathsf{saturate}(c(i) + g_c\,\mathsf{sgn}(\phi(i)), c_{\min}, c_{\max}) \\
v(i+1) &= \mathsf{saturate}(v(i) + g_v(c_{center} - c(i)), v_{\min}, v_{\max}) \\
\phi(i+1) &= \mathsf{wrap}(\phi(i) + (f_{dco}(c(i), v(i)) - f_{ref}) - g_\phi \phi(i)) \\
f_{dco}(c, v) &= \frac{1+\alpha v}{1+\beta c} f_0 \\
\mathsf{saturate}(x, lo, hi) &= \min(\max(x, lo), hi) \\
\mathsf{wrap}(\phi) &= \mathsf{wrap}(\phi + 1), \qquad \text{if } \phi \le -1 \\
&= \phi, \qquad \text{if } -1 < \phi < 1 \\
&= \mathsf{wrap}(\phi - 1), \qquad \text{if } 1 \le \phi
\end{aligned}
$$

The Convergence Proof[3]



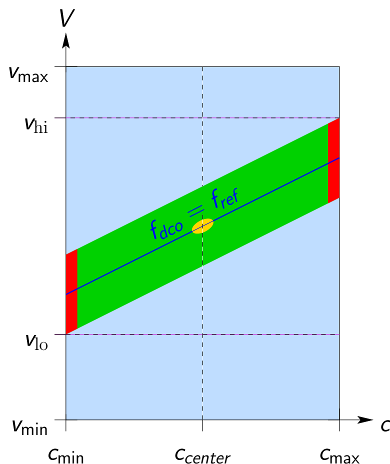
The global convergence property:

$$\exists N, \forall [c(0), v(0), \phi(0)] \in B,$$
$$\forall i \geq N, [c(i), v(i), \phi(i)] \in Y$$

In English: There exists a time bound such that for any initial state, the digital PLL reaches the final convergence region within that amount of time.

[3] Yan Peng and Mark Greenstreet. "Integrating SMT with Theorem Proving for Analog/Mixed-Signal Circuit Verification". In: *NASA Formal Methods*. 2015.

The Convergence Proof[3]

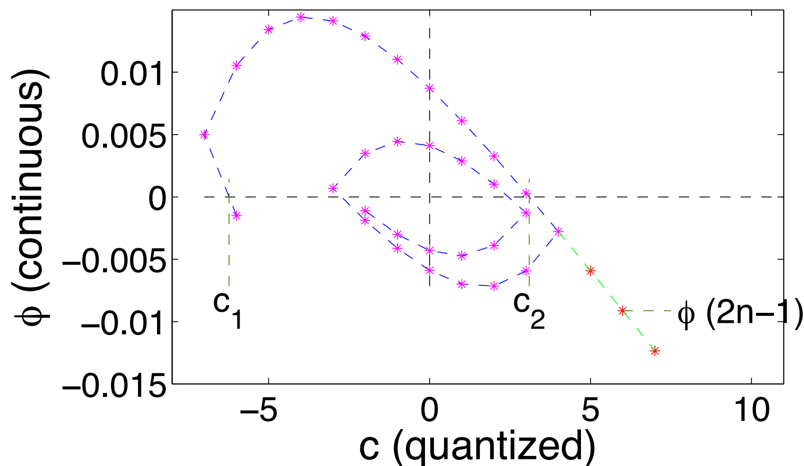


The convergence can be formulated using four lemmas:

- Coarse convergence: blue region to red and green region – Z3
- Leaving saturation: red region to green region – Z3
- Fine convergence: green region to yellow region – `Smtlink` 1.0
- Invariant: yellow region is invariant

[3] Yan Peng and Mark Greenstreet. "Integrating SMT with Theorem Proving for Analog/Mixed-Signal Circuit Verification". In: *NASA Formal Methods*. 2015.

Fine Convergence



- Requires reasoning about sequences of states.
- We proved that each crossing of $\phi = 0$ is closer to the origin than the previous one.

An example from the DPLL proof

Definitions:

$$\texttt{B-term(h)} = (1 - K_t)^{-h}(\mu \frac{1 + \alpha(d_0 + d_v)}{1 + \beta(g_1 h + (equ_c \ \ v_0))} - 1)$$

$$\texttt{B-sum(n)} = \sum_{h=1}^{n} (\texttt{B-term}(h) + \texttt{B-term}(-h))$$

An example from the DPLL proof

Key lemmas proved:

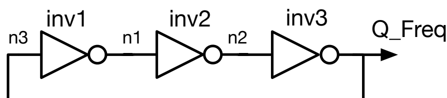
```
(defthm B-term-neg
  (implies (and (dpll-hyps :g1 :Kt :v0 :dv :pos h)
                (nc-ok h (- h)))
           (< (+ (B-term h v0 dv g1 Kt) (B-term (- h) v0 dv g1 Kt)
              ) 0))
  :hints (("Goal''"
           :smtlink-custom
           (:hypotheses (((implies (<= 2 h)
                                   (<= (expt (gamma Kt) h)
                                       (expt (gamma Kt) 2))))))))
  :rule-classes :linear)

(defthm B-sum-neg
  (implies (and (dpll-hyps :g1 :Kt :v0 :dv :pos n-minus-2)
                (nc-ok (- n-minus-2)))
           (< (B-sum n-minus-2 v0 dv g1 Kt) 0))
  :hints (("Goal" :in-theory (e/d (B-sum) (B-term)))))
```

1. Motivation
2. Smtlink
3. AMS Verification
4. Asynchronous Circuit Verification
5. Glitch Hunting
6. The Exciting Future Work

Motivation

1. Asynchronous design offers *many advantages*:
   1. It works when a design has more than one timing domain
   2. In some cases asynchronous designs can be faster or simpler than their synchronous counterparts
   3. Problem is naturally event-driven: neuromorphic chip design
2. But asynchronous circuits are more intellectually challenging to understand
3. The exact ordering of events is not statically determined; thus, asynchronous designs are also **non-deterministic**
4. We want to be able to verify *safety* and *liveness* properties of asynchronous circuits

The Simple Ring Oscillator Example

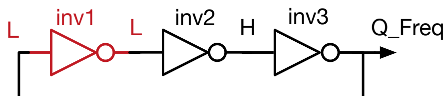


1. A ring oscillator is an oscillator circuit consisting of an odd number of inverters in a ring
2. A 3-stage ring oscillator consists of three inverters
3. The one-safe property:

Theorem (One-Safe)

Starting from a state where there is exactly one inverter ready-to-fire, for all future states, the ring oscillator will stay in a state where there is only one inverter ready-to-fire.

The Simple Ring Oscillator Example

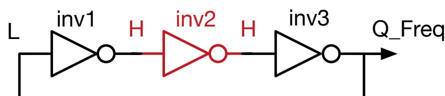


1. A ring oscillator is an oscillator circuit consisting of an odd number of inverters in a ring
2. A 3-stage ring oscillator consists of three inverters
3. The one-safe property:

Theorem (One-Safe)

Starting from a state where there is exactly one inverter ready-to-fire, for all future states, the ring oscillator will stay in a state where there is only one inverter ready-to-fire.

The Simple Ring Oscillator Example

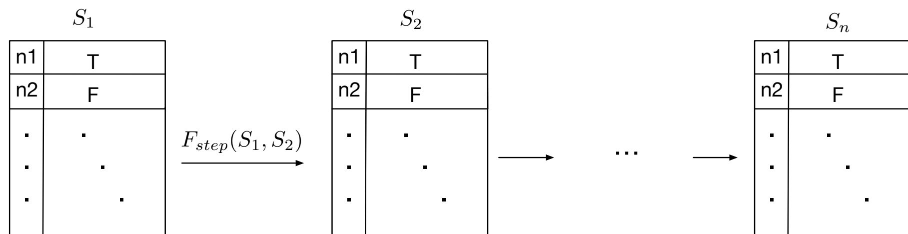


1. A ring oscillator is an oscillator circuit consisting of an odd number of inverters in a ring
2. A 3-stage ring oscillator consists of three inverters
3. The one-safe property:

Theorem (One-Safe)

Starting from a state where there is exactly one inverter ready-to-fire, for all future states, the ring oscillator will stay in a state where there is only one inverter ready-to-fire.

Modeling the Ring Oscillator



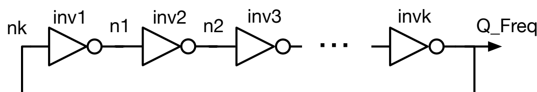
1. We model circuits using *trace recognizers* (based on [Dil87])
   1. A state is an alist mapping from signal paths to its state value
   2. A stepping function constrains possible next state; allows nondeterministic behaviors
   3. A trace is a list of states

The Theorem

```
(defthm ringosc3-one-safe
  (implies (and (ringosc3-p r) (any-trace-p tr) (consp tr)
                (ringosc3-valid r tr)
                (ringosc3-one-safe-state r (car tr)))
           (ringosc3-one-safe-trace r tr))
  :hints (("Goal"
           :induct (ringosc3-one-safe-trace r tr)
           :in-theory (e/d ...))
          ("Subgoal *1/1.1"
           :use ((:instance ringosc3-one-safe-lemma
                            (r r)
                            (tr tr)))
           )))
```

1. `ringoc3-one-safe-lemma`: the inductive step proved using `Smtlink`
2. `Smtlink` expands out definitions and z3 is able to derive enough relationships between terms to figure out the proof
3. `Smtlink` is very good at flattened formulas with large amount of details

Extend the Proof to Arbitrary Number of Stages

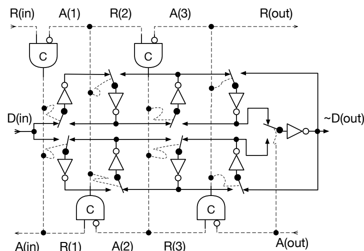


1. We've proven a theorem that states the one-safe property with a ring oscillator of arbitrary number of stages
2. Some statistics of the proof:

FTY types	Functions	Total thms	Smtlink thms	LOC
5	17	55	23	2375

3. Smtlink is smarter than I thought it was
4. There are still potential of improvements
    1. Much of the lengthiness of the proof is coming from having to expand terms out enough, so that `Smtlink` can handle the proof

The Micropipeline [Sut89]



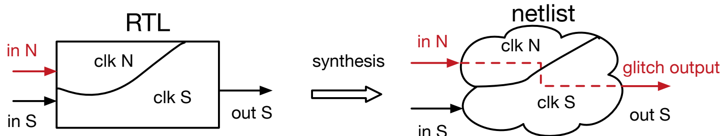
1. This is a timed-circuit: the correctness depends on the **inverters** propagating data values faster than the **C-elements** propagating control events
2. We've verified that the control path of a single stage micropipepline is one-safe
3. We plan to verify safety and functional correctness of the FIFO in the near future using trace theory as is used for the ring oscillator

1. Motivation
2. Smtlink
3. AMS Verification
4. Asynchronous Circuit Verification
5. Glitch Hunting
6. The Exciting Future Work

Motivation

1. Chip designs commonly contain multiple clock domains, multi-cycle paths, test circuits with long logic delays
2. Synthesis tools are based on circuit models that ignore the possibility that signals from other *clock domains can change at arbitrary times*
3. Glitch bugs are hard to find, nearly impossible in simulation
    1. Have to do frequency sweeping
4. This is a bug that has been found in real designs
5. The work shown here are published results in a paper in ASYNC2016 and a poster in DAC2018

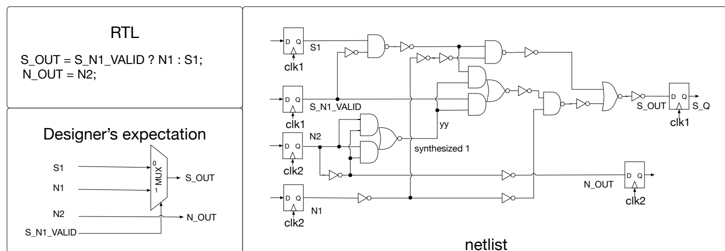
Synthesis-generated Glitch



Glitch: a transition on a non-synchronous signal can cause the output of the combinational logic to temporarily change to an unstable value.

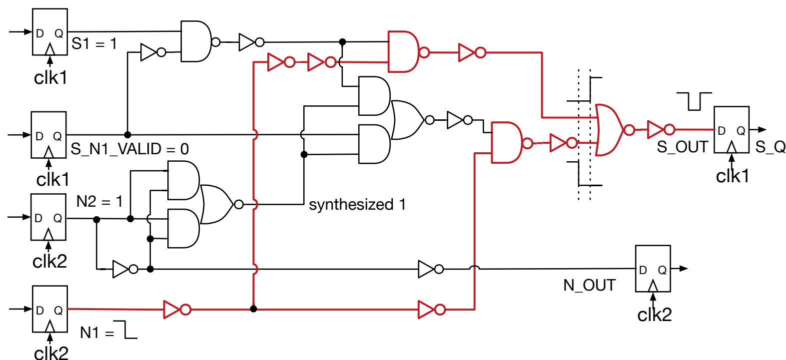
Synthesis-generated Glitch: synthesis tools can introduce glitches. This can happen even though the RTL design is free of such a glitch.

Is the netlist equivalent to the RTL?



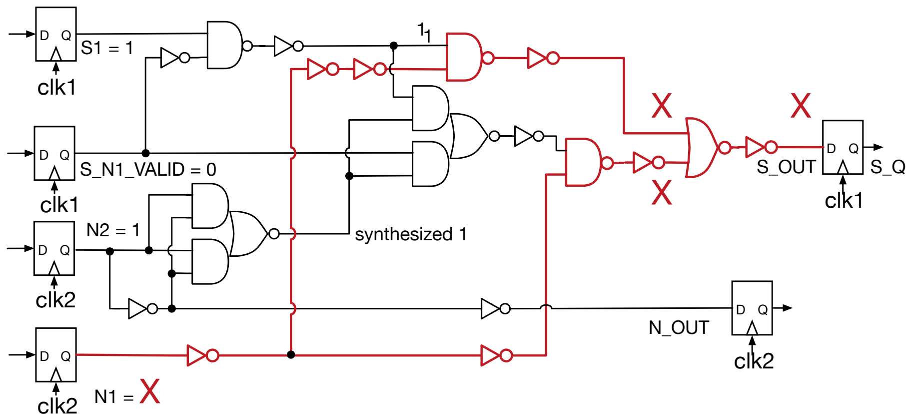
- YES! When using standard logical-equivalence checking
- Logical equivalence formulation:
  "For every input from {T, F}, the netlist produces the same output as the RTL."
- Signal naming:
  - S – signals Synchronous to output clock domain
  - N – signals Non-synchronous to output clock domain

Glitches caused by non-synchronous signals



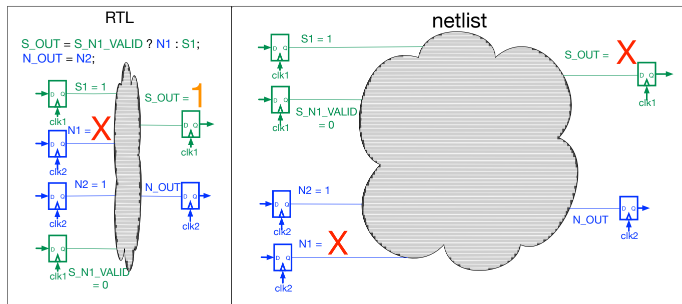
- Standard logical-equivalence is not enough, e.g., when `S_N1_VALID` is 0:
  - RTL: permits only `S1` to pass to the MUX output, `S_OUT`
  - netlist: allows a glitch to propagate from `N1` to `S_OUT`

Using ternary simulation to detect glitch



- Ternary logic values {T, F, X} facilitate detection of glitch paths

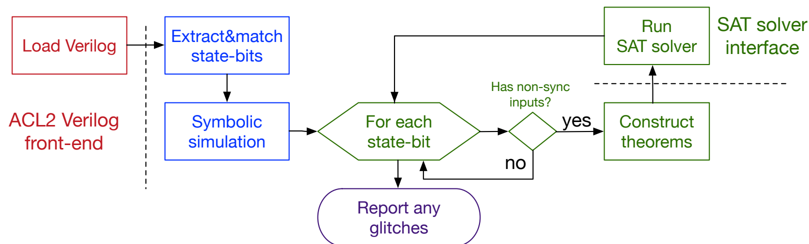
The Formal Definition



- For a state-bit, q, let $\mathcal{S}_q$ denote the synchronous inputs to the combinational logic for the next-state of q, and $\mathcal{N}_q$ denote the non-synchronous inputs. Let $\mathbb{B} = \{0, 1\}$, and $\mathbb{B}^{\mathbf{X}} = \{0, 1, \mathbf{X}\}$

$$\begin{aligned} \text{glitchFree(q)} \;=\; & \forall \mathcal{S}_q \in \mathbb{B}^*.\ \forall \mathcal{N}_q \in \mathbb{B}^{\mathbf{X}*}. \\ & (\text{next}_{q,\text{net}}(\mathcal{S}_q, \mathcal{N}_q) = \mathbf{X}) \Rightarrow (\text{next}_{q,\text{RTL}}(\mathcal{S}_q, \mathcal{N}_q) = \mathbf{X}) \end{aligned} \quad (1)$$

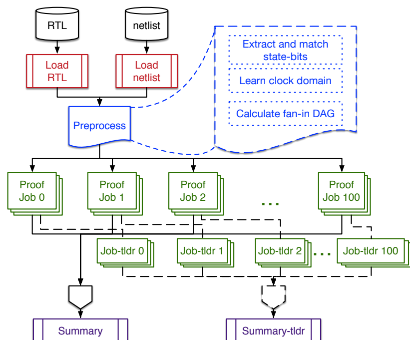
Sequential Glitch Hunter[4]



1. ACL2 provides a comprehensive Verilog front end and a SAT solver interface
2. Theorems are automatically constructed and proved for all statebits
3. When a glitch is found, a counter-example is shown indicating the glitch inputs

[4] Y. Peng, I. W. Jones, and M. R. Greenstreet. "Finding Glitches Using Formal Methods". In: *2016 22nd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. 2016.

Parallel Glitch Hunter



1. distribute computation over multiple machines by leveraging the ACL2 certification method and the Unix Make utility
2. Fault-tolerant parallel runs
3. Other performance improvements: fast-alists (i.e. applicative maps backed by hash tables), memoization and guards

Experimental Results

Table: Modules and Run Time

Module	#gates	#FFs	#GH-FFs[a]	T_{32}[b]	T_{min}[c]	P_{min}[d]
Module A	1264	721	221(30.7%)	6	6	16
Module B	10923	4256	2378(55.9%)	17	13	96
Module C	90432	14874	2045(13.7%)	22	20	96
Module D	29018	5092	2293(45.0%)	84	84	32
Module E	238783	177996	53415(30.0%)	446	280	100

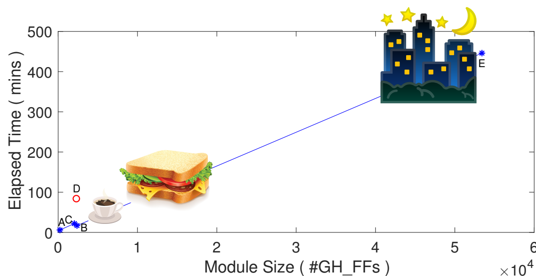
[a] GH-FFs are state-bits that include non-synchronous inputs in their fan-in trees

[b] Time (in minutes) with 32 parallel jobs

[c] T_{min} is the fastest run (in minutes) for the module

[d] P_{min} is the number of processors for the fastest run

Experimental Results – Cont'd



1. For modules with a few thousand gates, the time to dispatch jobs dominates, and 16 or 32 processors seems optimal
2. For modules with hundreds of thousands of gates, the preprocessing step is a sequential bottleneck accounting for about 2% of the total computation and limiting speed-up to around 50
3. Outlier module D for preprocessing time due to combinational loops

1. Motivation
2. Smtlink
3. AMS Verification
4. Asynchronous Circuit Verification
5. Glitch Hunting
6. The Exciting Future Work

The Exciting Future Work

- Short term:
  1. For `Smtlink`, we want to add reflection and type inference
  2. We are interested in applying `Smtlink` to several asynchronous designs
     1. Formally verify Sutherland's micropipepline
     2. The verification of an initialization problem of an deskew ASP* FIFO
- Long term: I'm very interested in applying `Smtlink` to software problems.
  1. Distributed systems: bares similarities to asynchronous circuits
  2. Machine learning algorithms: convergence of various versions of stochastic gradient problems

Conclusion

Conclusion: we showed how combining theorem proving and SMT/SAT solving have great potential in verifying timed and continuous systems.

1. We have built a novel connection of SMT solvers into the theorem prover ACL2, called `Smtlink`
2. We verified convergence of a digital Phase-Locked Loop and plan to reason about asynchronous circuits in the near future using `Smtlink`
3. During my internship at Oracle, I applied a similar idea of using the ACL2 theorem prover and SAT solvers for solving an industry problem of detecting synthesis-generated glitches

Questions?

Maybe you should consider asking `Smtlink` that question? ...

ACL2::projects

Smtlink

[books]/projects/smtlink/doc.lisp

SMT Package

Tutorial and documentation for the ACL2 book, Smtlink.

Introduction

A framework for integrating external SMT solvers into ACL2 based on the ACL2::clause-processor and the `ACL2::computed-hints` mechanism.

Overview

`Smtlink` is a framework for representing suitable ACL2 theorems as a SMT (Satisfiability Modulo Theories) formula, and calling SMT solvers from within ACL2.

References I

J. Crossley, E. Naviasky, and E. Alon. "An energy-efficient ring-oscillator digital PLL". In: *Custom Integrated Circuits Conference (CICC), 2010 IEEE*. 2010, pp. 1–4. DOI: 10.1109/CICC.2010.5617417.

David L. Dill. "Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits". AAI8814716. PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University, 1987. URL: http://reports-archive.adm.cs.cmu.edu/anon/scan/CMU-CS-88-119.pdf.

Yan Peng and Mark Greenstreet. "Integrating SMT with Theorem Proving for Analog/Mixed-Signal Circuit Verification". In: *NASA Formal Methods*. 2015.

References II

Y. Peng and M. R. Greenstreet. “Smtlink 2.0”. In: *15th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2018)*. 2018.

Y. Peng, I. W. Jones, and M. R. Greenstreet. “Finding Glitches Using Formal Methods”. In: *2016 22nd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. 2016.

I. E. Sutherland. “Micropipelines”. In: *Commun. ACM* 32.6 (June 1989), pp. 720–738. ISSN: 0001-0782. DOI: 10.1145/63526.63532. URL: http://doi.acm.org/10.1145/63526.63532.

References III

J. Wei et al. “Verifying global convergence for a digital phase-locked loop”. In: *2013 Formal Methods in Computer-Aided Design*. 2013, pp. 113–120. DOI: 10.1109/FMCAD.2013.6679399.

There are Always Exceptions - Precondition Example

```
(fty::deflist intlist
  :elt-type integerp
  :true-listp t)

(defthm bogus
  (implies (intlist-p x)
           (or (< (car x) 0)
               (equal (car x) 0)
               (> (car x) 0))))
```

$x =$ `nil` is a counter-example to this bogus theorem:

let $x =$ `nil`:

`(or (< (car nil) 0) (equal (car nil) 0) (> (car nil) 0))`

`(car nil)` $=$ `nil`:

`(or (< nil 0) (equal nil 0) (> nil 0))`

All comparisons of non-numbers produce `nil`:

`(or nil nil nil)` $=$ `nil`

Precondition Example Cont'd.

A direct translation of the ACL2 goal:

```
IntList = Datatype('IntList')
IntList.declare('cons', ('car', IntSort()),
                        ('cdr', IntList))
IntList.declare('nil')
IntList = IntList.create()

x = Const('x', IntList)
prove(Or(IntList.car(x) > 0, IntList.car(x) == 0,
    IntList.car(x) < 0))
```

But $x =$ `nil` is not a counter-example to this Z3 theorem. Because `IntList.car(nil)` in Z3 denotes an arbitrary integer value, and the theorem trivially holds.

Precondition Example Cont'd.

The problem:

- ACL2: Taking `car` of `nil` gives us `nil`
- Z3: Taking `car` gives us an arbitrary value of the appropriate type

Solution: add precondition check $x \neq$ `nil` in places where `(car x)` is applied;
Similarly, for `(cdr (assoc-equal key alist))`, precondition check `(assoc-equal key alist)` $\neq$ `nil`