Integrating SMT with Theorem Proving for AMS Verification

Yan Peng & Mark Greenstreet

University of British Columbia
Vancouver, CA

July 09, 2014

Outline

Integrating SMT with Theorem Proving for AMS Verification

✿ **Contributions**

- Integrating SMT with Theorem Proving, challenges and solutions
- Verifying global convergence of a Digital Phase-Locked Loop(DPLL) using recurrence
- Conclusion

Contributions

- Combine industrial strength SMT solver with industrial strength theorem prover.
- Model state-of-the-art DPLL with recurrences.
- Proof of global convergence.
- Able to prove design with parameter variation.

Outline

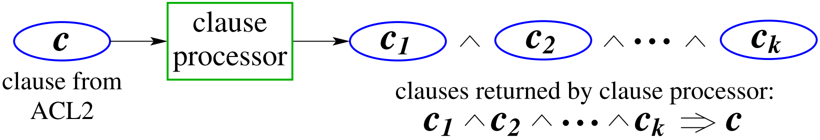
Integrating SMT with Theorem Proving for AMS Verification

- Contributions
- **Integrating SMT with Theorem Proving**
  - **Why combine Z3 and ACL2?**
  - **Software framework and technical challenges**
- Verifying global convergence of a Digital Phase-Locked Loop(DPLL) using recurrence
- Conclusion

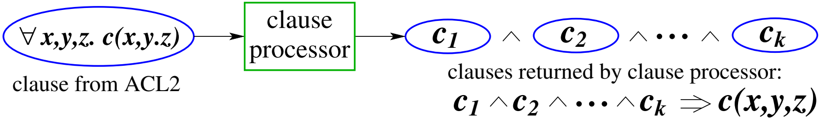
- *Satisfiability Modulo Theories (SMT)* problem is a unified decision procedure for logical formulas which combines solvers for a rich set of background theories.
- Possible theories: propositional logic, arithmetic, uninterpreted functions, bitvectors theories etc.
- *Z3, Microsoft Research* [MB08, JM12]. Non-linear arithmetic theories, suitable for AMS design with non-linear dynamics.
- Lack of:
  - Induction proof
  - Structured proof

- *Theorem proving* is a technique for proving a set of theorems by building upon a set of basic axioms and use of logic rules, e.g. rewrite rules, induction.
- In order to prove a final theorem, one looks at what is needed and develops a set of lemmas.
- ACL2, University of Texas at Austin.[KM97]
- But working through complicated boolean formulas, systems of inequalites, etc., can be extremely tedious.
- ACL2 and Z3 complement each other:
  - ACL2 provides structured proofs and induction proofs.
  - Z3 discharges complicated/tedious systems of inequalities.

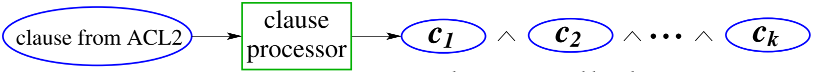
SMT and Theorem Proving - clause processor



- A clause processor takes the goal one wants to prove and decomposes the goal into a conjunction of subgoals. Each subgoal is a called a clause.
- ACL2 supports two kinds of clause processors:
  - A **verified** clause processor is written in Lisp and proven correct within ACL2.
  - A **trusted** clause processor is anything else.
    Theorems whose proofs rely on a trusted clause processor are tagged accordingly.
- We integrate Z3 into ACL2 as a trusted clause processor.



- Challenge: ACL2 has rationals and Z3 has reals.
  - In ACL2, $\neg \exists x.\ x^2 = 2$ is a theorem.
  - In Z3, $\exists x.\ x^2 = 2$ is a theorem.
- Solution: only use Z3 to prove propositions where all variables are universally quantified.



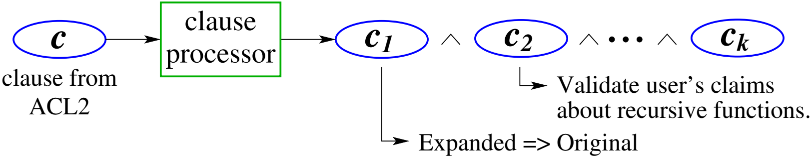

```
(implies (and (rationalp x)
              (rationalp y)
              (rationalp z))
         (c x y z))
```

clauses returned by clause processor:

$$c_1 \wedge c_2 \wedge \cdots \wedge c_k \Rightarrow \mathbf{c(x,y,z)}$$

- Challenge: ACL2 is untyped but Z3 is typed.
- Solution: user adds type assertions to antecedent.
  - These are almost always needed anyways.
  - This requirement is not a significant burden.

Challenge: user defined functions



- Challenge:
  - ACL2 supports arbitrary lisp functions.
  - Z3 functions are more like macros (no recursion).
- Solution:
  - Set up translation for a basic set of functions.
  - Expand non-recursive functions.
  - Expand recursive functions to bounded depth.
  - Expansion done on ACL2's representation: can verify correctness.

Other issues:

- Claims can contain non-polynomial terms.
  - Replace offensive subexpression with a variable.
  - User adds constraints about the variable.
  - These constraints are returned as clauses for ACL2 to prove.
- ACL2 may need hints to discharge clauses returned from the clause processor.
  - Solution: nested hints.
  - These hints tell the clause processor what hints to attach to returned clauses.
- These features provides a very flexible back-and-forth between induction proofs in ACL2 and handling the details of the algebra with Z3.

Example - the theorem

$$\forall\ a\ b\ \gamma \in R, m\ n \in Z. If\ 0 < m < n,\ 0 < gamma < 1. \rightarrow$$
$$\gamma^m((a+b)^2 - 2ab) \geq \gamma^n \cdot 2ab$$

```
(defun f-mul-2 (x) (f-mul 2 x))
(defun f-plus (x y) (+ x y))
(defun f-square (x) (f-mul x x))
(defun f-neg (x) (- x))
(defun f-minus (x y) (f-plus x (f-neg y)))
(defun f-expt (x n) (expt x n))
```

Example - code

```
(defthm demonstration
  (implies (and (and (rationalp a)
                     (rationalp b)
                     (rationalp gamma)
                     (integerp m)
                     (integerp n))
                (and (> gamma 0)
                     (< gamma 1)
                     (> m 0)
                     (< m n)))
           (>= (f-mul (expt gamma m)
                      (f-minus (f-square (f-plus a b))
                               (f-mul (f-mul-2 a) b)))
               (f-mul (foo gamma n)
                      (f-mul (f-mul-2 a) b))))
  :hints ...)
```

Example - code

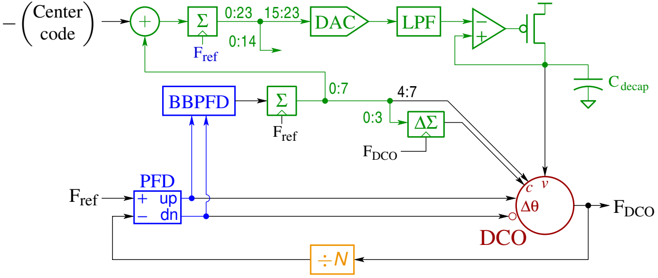
```
:hints
(("Goal"
  :clause-processor
  (my-clause-processor clause
    '( (:expand ((:functions ((f-mul rationalp)
                              (f-mul-2 rationalp)
                              (f-plus rationalp)
                              (f-square rationalp)
                              (f-neg rationalp)
                              (f-minus rationalp)
                              (f-expt rationalp)))
                 (:expansion-level 1))
                (:python-file "demonstration")
                (:let ((expt_gamma_m (expt gamma m) rationalp)
                       (expt_gamma_n (expt gamma n) rationalp)))
                (:hypothesize ((< expt_gamma_n expt_gamma_m)
                               (> expt_gamma_m 0)
                               (> expt_gamma_n 0)))
                (:use ((:type ())
                       (:hypo ())
                       (:main ())))))))))
```

Outline

Integrating SMT with Theorem Proving for AMS Verification

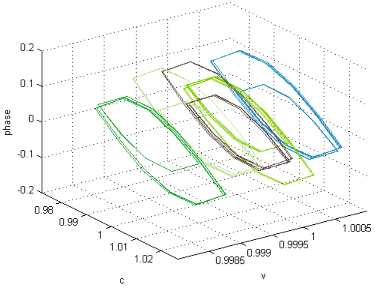
- Contributions
- Integrating SMT with Theorem Proving
- **Verifying global convergence of a Digital Phase-Locked Loop(DPLL) using recurrence**
  - **The state-of-the-art Digital PLL**
  - **Establish recurrence model for the DPLL**
  - **Prove global convergence using Z3 and ACL2**
- Conclusion

A state-of-the-art Digital PLL (from CICC 2010)[CNA10]



- DCO has three control inputs:
  capacitance setting (digital), supply voltage (linear), phase correction (time-difference of digital transitions).
- Uses linear and bang-bang PFD.
- Integrators are digital.
- LPF and decap to improve power-supply rejection.
- It is impractical to verify global convergence using simulation.

Establish the Recurrence Model



A limit cycle is an isolated closed trajectory, for which its neighbouring trajectories are not closed they spiral either towards or away from the limit cycle.

- The recurrence model:

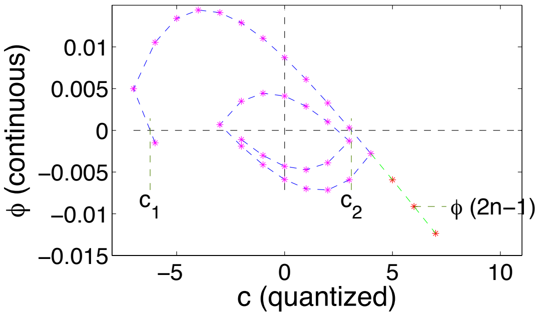
$$\begin{aligned} c(i+1) &= c(i) + g_1 sign(\phi(i)) \\ v(i+1) &= v(i) + g_2(c(i) - c_{code}) \\ \phi(i+1) &= (1-K_t)\phi(i) + 2\pi\left(\frac{f_{dco}(i)}{Nf_{ref}} - 1\right) \end{aligned}$$

where $f_{dco}(i) = f_0 \frac{1+\alpha v(i)}{1+\beta c(i)}$

The proof: the big picture

- Coarse convergence: from any initial condition, ϕ eventually crosses 0 in a state where *c* and *v* are not saturated.
  - Proof sketch:
  - Use Ricatti equation to get a ranking function based on linear model at convergence.
  - Use this ranking function to show coarse convergence using non-linear, global model.
  - Z3 discharges all of the proof obligations.
- Fine convergence: from any crossing of $\phi = 0$ with *c* and *v* away from their saturation conditions (as established above), ϕ will continue to make zero-crossings that each move closer to the intended equilibrium.
  - Proof sketch: see the next few slides.

The proof: fine convergence using induction proof



- Solve the recurrence (verified by ACL2 – rewrite & induction):

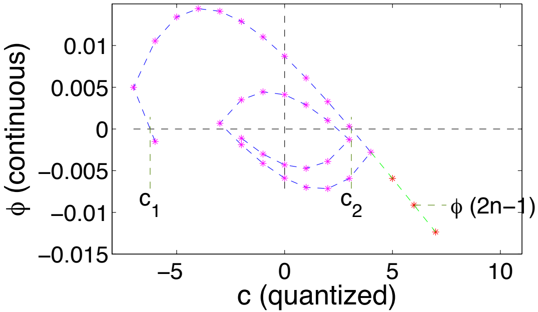
$$\begin{aligned} c(j) &= c_0 + g_1 j \\ \phi(j) &= \gamma^j \phi_0 + 2\pi \sum_{i=0}^{j-1} \gamma^{(j-1-i)} \left(\mu \frac{1+\alpha v}{1+\beta c(i)} - 1 \right) \end{aligned}$$

- We want to prove:

$$\forall m \geq 3,\ \phi(2m-1) < 0$$

A symmetric argument applies to lower half of the space.

The proof



- Exploit the asymmetry between terms with $c < c_{eq}$ and $c > c_{eq}$ where c_{eq} is chosen to set $f_{dco} = f_{ref}$.
  - We pair up points to simplify the formula.
  - Basic idea: the negative terms dominate the positive ones.
- Proving these claims manually involves many pages of messy algebra:
  - Just have Z3 takes care of it.

Conclusion

- We've shown an integration of the Z3 SMT solver into the ACL2 theorem prover with applications for AMS verification.
- Theorem proving is hard! Reachability is easy! Why use a theorem prover?
  - Reachability tools only solve parts of the problem. Human reasoning is needed to conclude that the system works given these partial results.
  - Our formulation lets us work directly on the recurrences rather than on continuizations:
    - Can reason in detail about limit-cycle behaviour.
  - We hope for "re-usable proofs."
    - Proof re-use has been very useful in the hardware verification world.
    - AMS verification seems amenable to the same approach: there aren't that many types of AMS blocks even though there are many implementations.

References

J. Crossley, E. Naviasky, and E. Alon, *An energy-efficient ring-oscillator digital pll*, Custom Integrated Circuits Conference (CICC), 2010 IEEE, 2010, pp. 1–4.

Dejan Jovanović and Leonardo Moura, *Solving non-linear arithmetic*, 6th International Conference on Automated Reasoning (Bernhard Gramlich, Dale Miller, and Uli Sattler, eds.), Lecture Notes in Computer Science, vol. 7364, Springer Berlin Heidelberg, June 2012, pp. 339–354.

Matt Kaufmann and J. S. Moore, *An industrial strength theorem prover for a logic based on common lisp*, IEEE Trans. Softw. Eng. **23** (1997), no. 4, 203–213.

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.