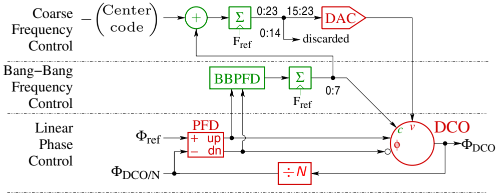
# Smtlink 2.0

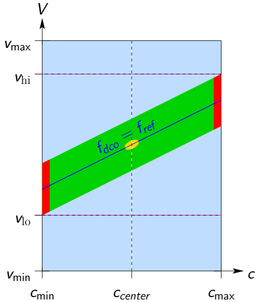
Yan Peng[1] Mark R. Greenstreet[1]

[1]Department of Computer Science
University of British Columbia

November 6th 2018

# Smtlink 1.0

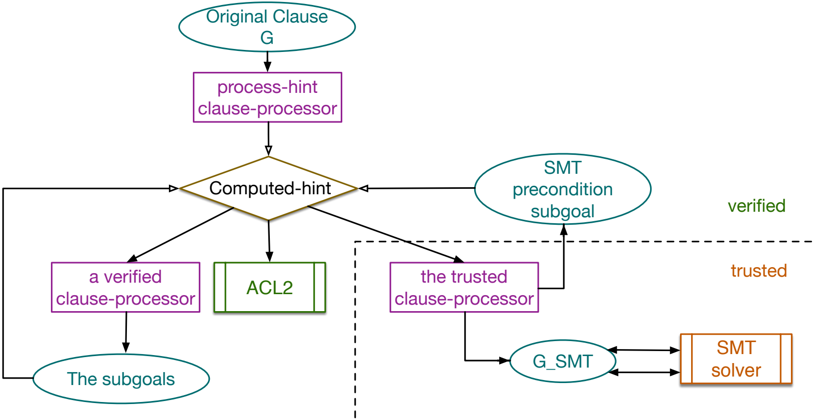



1. Achievement: `Smtlink`'s supports for *linear and non-linear arithmetics of integers and rationals* helps forming the DPLL global convergence proof
2. Limitations: thought of as only useful when it comes to problems involving non-linear arithmetics
3. But, `Smtlink` should be **more than that**.

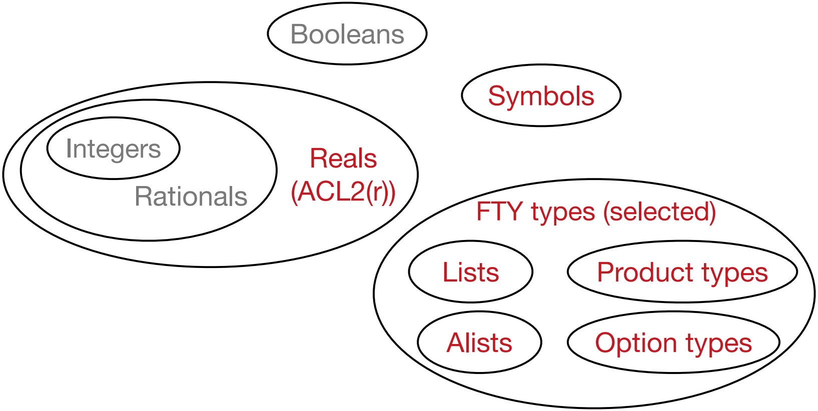
# What's New in Smtlink 2.0

1. An extensible architecture



# What's New in Smtlink 2.0

1. An extensible architecture
2. A richer support of datatypes



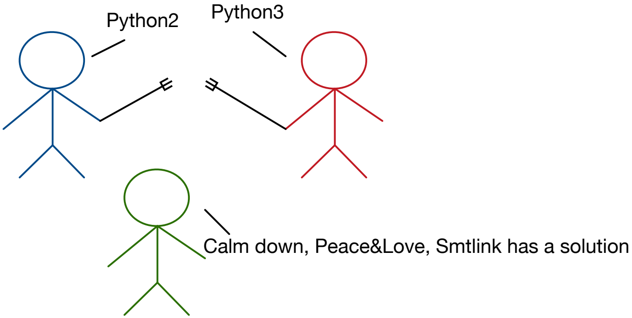
# What's New in Smtlink 2.0

1. An extensible architecture
2. A richer support of datatypes
3. Better user interface: follows the `define` convention and the `:hints` convention

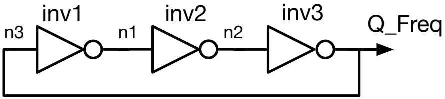
```
:hints(("Goal"
        :smtlink
        (:functions ((foo :formals ((x real/rationalp))
                          :returns ((rx real/rationalp))
                          :level 0))
         :hypotheses (((<= 1 (foo x))
                       :hints
                       (:use ((:instance foo->=-1
                                         (x x))))))
        )))
```

# What's New in Smtlink 2.0

1. An extensible architecture
2. A richer support of datatypes
3. Better user interface: follows the `define` convention and the `:hints` convention
4. Now supports both Python 2 and Python 3



# The Simple Ring Oscillator Example

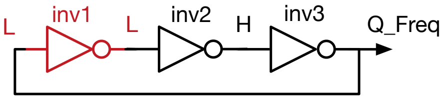



1. A ring oscillator is an oscillator circuit consisting of an odd number of inverters in a ring
2. A 3-stage ring oscillator consists of three inverters
3. The one-safe property:

**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

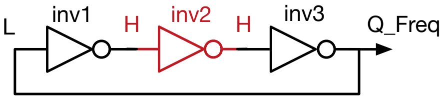



1. A ring oscillator is an oscillator circuit consisting of an odd number of inverters in a ring
2. A 3-stage ring oscillator consists of three inverters
3. The one-safe property:

**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

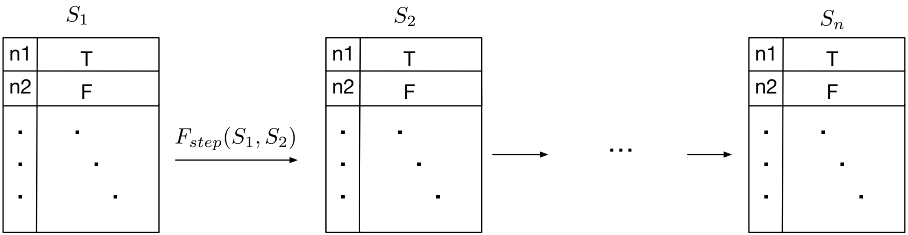



1. A ring oscillator is an oscillator circuit consisting of an odd number of inverters in a ring
2. A 3-stage ring oscillator consists of three inverters
3. The one-safe property:

**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



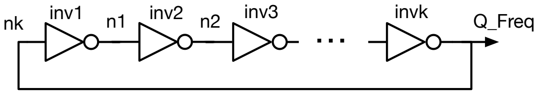

1. We model circuits using *trace recognizers* (based on [Dil87])
    1. A state is an alist mapping from signal paths to its state value
    2. A stepping function constrains possible next state; allows nondeterministic behaviors
    3. A trace is a list of states

# 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



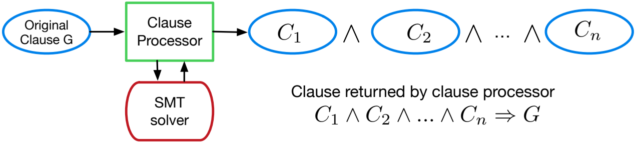

1. We've proven a theorem that states the one-safe property with a ring oscillator of arbitrary number of stages
2. Some statistics of the proof:

| 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 Story for a New Architecture

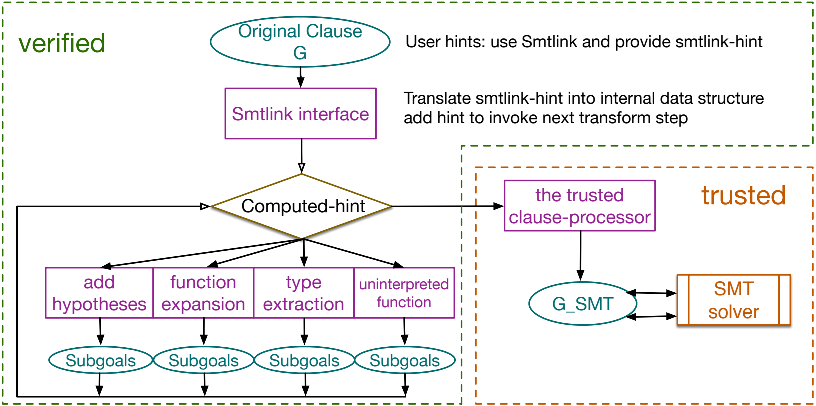
1. The old architecture is monolithic: one single trusted clause-processor




2. After the 2015 workshop, based on Jared's suggestions, Matt, Dave, Dmitry, Mark and I discussed the possibility of using computed-hint. Lead to the file: books/hints/hint-wrapper.lisp
3. The idea is to use a verified clause-processor that generates multiple clauses, and put markers on clauses that can be recognized by computed-hints for further steps
4. This further leads to the new architecture

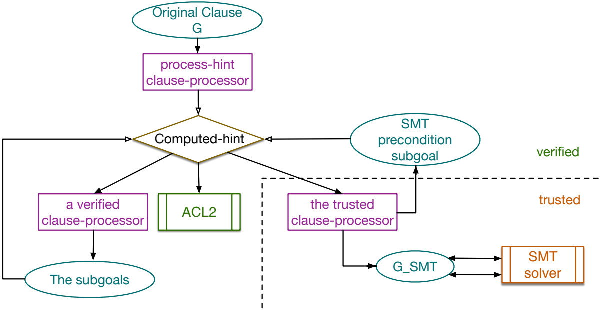
# The Architecture

**The new architecture is both extensible and has a more 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

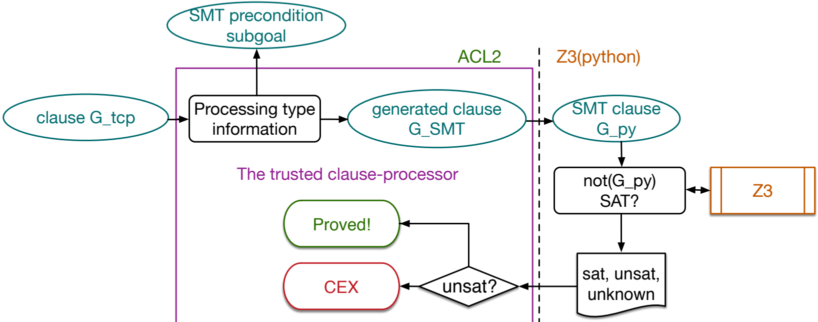



*smt-architecture* table

| 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.

# 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`

# 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 $k^{th}$ 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

# The Exciting Future Work

1. Types are crucial to using SMT solvers, need a type inference engine
2. Reflection allowed by meta-extract: removes the necessity of proving auxiliary theorems. We plan to add:
   1. Verified function expansion
   2. Verified type inference
3. More induction proof support
4. Fully evaluable counter-examples

# Conclusion

Conclusion: We built a new version of Smtlink that has a more compelling argument for soundness, has an extensible architecture and is more user-friendly.

1. How can I start using it?

```
(include-book "projects/smtlink/top" :dir :system)
(value-triple (tshell-ensure))
(add-default-hints '((SMT::SMT-computed-hint clause)))
```

2. Documentation: `:doc smtlink` or go to XDOC website
3. Smtlink is under active development right now. We're eager to hear any feedback!

# 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


David L. Dill.

*Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*.

PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1987.

AAI8814716.