# GLMC

Connecting ACL2 with Hardware Model Checkers

# Proving Invariants in Hardware Verification

- Inductive invariants are “easy” to prove
    - Provable by SAT for finite state machines
    - In ACL2, can use GL.
- Downsides:
    - Hard to find
    - Brittle, implementation-sensitive
- Model-checking proves invariants that aren’t necessarily inductive
    - Automatically searches for inductive invariant that implies the invariant you want.
    - Increasingly powerful algorithms: explicit state → BDDs → interpolation → PDR/IC3
    - Available in open source tools, e.g. ABC

## GLMC Operation

- (User): Break down the problem.  Parts:
  - Frame inputs
  - Next-state function
  - Invariant property
  - Initial state predicate
  - Constraints
- (GLMC): Express everything as Boolean functions
  - AIG representation
- Solve using external model checker
  - Configurable by attachment
  - ABC is a suitable open-source one
  - Or write one in ACL2 (and release it, please!)

## Very Simple Example

- Machine counts up modulo 10
- Inputs: reset, increment
- Want to know: never reaches 14
    - Not an inductive invariant!

```
(defun my-nextst (st incr reset)
  (b* (((when reset) 0)
       (st (lnfix st))
       ((unless incr) st)
       (next (1+ st))
       ((when (eql next 10)) 0))
    next))

(defund my-run-prop (st ins)
  (declare (xargs :measure (len ins)))
  (if (atom ins)
      t
    (and (not (equal st 14))
         (my-run-prop (my-nextst st (caar ins) (cdar ins)) (cdr ins)))))

(defthm my-run-prop-correct
  (implies (and (natp st)
                (< st 5))
           (my-run-prop st ins)))  ;; Not inductive!
```

```
(defthm my-run-prop-correct
  (implies (and (natp st)
                (< st 5))
           (my-run-prop st ins))
  :hints ((glmc-hint
           :shape-spec-bindings `((incr ,(g-var 'incr))
                                  (reset ,(g-var 'reset))
                                  (st ,(g-int 2 1 5)))
           :state-var st
           :initstatep (< st 5)
           :nextstate (my-nextst st incr reset)
           :frame-input-bindings ((incr (caar ins))
                                  (reset (cdar ins)))
           :rest-of-input-bindings ((ins (cdr ins)))
           :end-of-inputsp (atom ins)
           :measure (len ins)
           :run (my-run-prop st ins)
           :state-hyp (and (natp st) (< st 16))
           :prop (not (equal st 14))
           :run-check-hints ('(:expand ((my-run-prop st ins))))
```

# Hardware Model-checking with GLMC

(Experimental!)

```
module counter (input clk,
                input reset,
                input incr,
                output logic [3:0] count);

    always @(posedge clk) begin
      automatic logic [3:0] tmpcount = count;
      if (reset) begin
        tmpcount = 0;
      end else begin
        tmpcount = tmpcount + incr;
      end
      if (tmpcount == 10)
        tmpcount = 0;
      count <= tmpcount;
     end

endmodule
```

```
(defsvtv counter-step
  :mod *counter*
  :inputs  '(("clk" 0  1)
             ("reset"  reset _)
             ("incr"   incr  _))
  :outputs '(("count" count _))
  :state-machine t)
```

```
(define counter-run-step ((ins svex-env-p)
                          (st svex-env-p))

  (b* (((svtv counter) (counter-step))
       (ins (make-fast-alist ins))
       ((mv (list step) (list nextst))
        (svtv-fsm-run-outs-and-states
          (list ins) st (counter-step)
          :out-signals '((count reset incr))
          :state-signals (list (alist-keys counter.nextstate)))))

    (mv (make-fast-alist step)
        (make-fast-alist nextst))))
```

```
(define counter-ok ((st svex-env-p)
                    (ins svex-envlist-p))
  (b* (((when (atom ins)) t)
       ((svtv counter) (counter-step))
       (in (car ins))
       ((mv step nextst) (counter-run-step in st))
       (count (svex-env-lookup 'count step))
       (reset (4vec-zero-ext 1 (svex-env-lookup 'reset in)))
       (incr (4vec-zero-ext 1 (svex-env-lookup 'incr in)))
       ((unless (and (2vec-p reset)
                     (2vec-p incr))) t)
       ((unless (and (2vec-p count)
                     (not (equal (2vec->val count) 14))))
        nil))
    (counter-ok nextst (cdr ins))))
```

```
(defthm counter-is-ok
  (b* ((((mv step &) (counter-run-step (car ins) st))
        (count (svex-env-lookup 'count step)))
    (implies (and (2vec-p count)
                  (< count 5))
             (counter-ok st ins)))
  :hints ((gl::glmc-hint
           :state-var st
           :nextstate (b* (((mv & nextst) (counter-run-step in st)))
                        nextst)
           :prop (b* (((mv step &) (counter-run-step in st))
                      (count (svex-env-lookup 'count step)))
                   (and (2vec-p count)
                        (not (equal (2vec->val count) 14))))
           :constraint (and (2vec-p (4vec-zero-ext 1 (svex-env-lookup 'reset in)))
                            (2vec-p (4vec-zero-ext 1 (svex-env-lookup 'incr in))))
           :initstatep (b* (((mv step &) (counter-run-step in st))
                            (count (svex-env-lookup 'count step)))
                         (and (2vec-p count)
                              (< count 5)))
           :frame-input-bindings ((in (car ins)))
           :rest-of-input-bindings ((ins (cdr ins)))
           :end-of-inputsp (atom ins)
           :measure (len ins)
           :run (counter-ok st ins)
           :shape-spec-bindings `((in ,(gl::g-var 'in))
                                  (st ,(gl::g-var 'st)))
           :run-check-hints ('(:expand ((counter-ok st ins)))))))
```

# Questions?

- Released soon
- Yes, the interface is baroque
- Generates counterexamples
- Works with GL term-level stuff
- Performance mostly depends on backend model checker