Proving Invariants via Rewriting and Finite Search

ACL2 Seminar

August 18, 2004

Rob Sumners

robert.sumners@amd.com

⌊ **What are *Invariants*?** ⌋

- A *Term* is either a variable symbol, a quoted constant, or a function application
  - Example:
    `(cons (binary-+ x (quote 1)) '(t . nil))`
    - Every *function* is either a function symbol or a `lambda` expression
- A *Predicate* is a term with a single variable symbol `n` and is interpreted in an `iff` context
  - This is our non-standard definition of *Predicate*
- An *Invariant* is a predicate which we wish to prove is non-`nil` for all values of `n`.
  - The variable `n` is intended to range over all values of natural-valued "time"

⌊ Importance of Proving Invariants ⌋

- Caution – over-generalized statement which I do not wish to debate:
  - Most properties of interest about concurrent, reactive systems can be effectively reduced to the proof of a sufficient invariant
- Invariants can be very difficult and tedious to prove for larger systems.
  - Many prime examples of this from our community and other formal methods communities
  - From ACL2 community: CLI stack work, Jun's work, Pete's work, JVM work, Sandip's work, My work, etc.

⌊ Example Invariant: Mutual Exclusion ⌋

```
(encapsulate (((i *) => *))
  (local (defun i (n) n)))

(define-system mutual-exclusion

 (in-critical (n) nil
   (if (in-critical n-)
       (/= (i n) (critical-id n-))
     (= (status (i n) n-) :try)))

 (critical-id (n) nil
   (if (and (not (in-critical n-))
            (= (status (i n) n-) :try))
       (i n)
     (critical-id n-)))

 (status (p n) :idle
   (if (/= (i n) p) (status p n-)
     (case (status p n-)
           (:try (if (in-critical n-)
                     :try
                   :critical))
           (:critical :idle)
           (t :try)))))
```

⌊ Specifying Mutual Exclusion ⌋

• Property: No two distinct processes a and b can be in the `:critical` state at the same time

• Codified as the invariant `(ok n)`:

```
(encapsulate (((a) => *) ((b) => *))
  (local (defun a () 1))
  (local (defun b () 2))
  (defthm a-/=-b (not (equal (a) (b))))
  (defthm a-non-nil (not (equal (a) nil)))
  (defthm b-non-nil (not (equal (b) nil))))

(defun ok (n)
  (not (and (= (status (a) n) :critical)
            (= (status (b) n) :critical))))
```

⌊ Approaches - Theorem Proving ⌋

- Define and prove an *inductive invariant* which implies the target invariant.

  – For complex systems, the definition and/or proof of an inductive invariant is a non-trivial exercise

- For our mutual exclusion example:

```
(defun ii-ok-for1 (n i)
  (iff (= (status i n) :critical)
       (and (in-critical n)
            (= (critical-id n) i))))

(defun ii-ok (n)
  (and (ii-ok-for1 n (a)) (ii-ok-for1 n (b))))

(defthm ok-is-invariant
  (and (ii-ok 0)
       (implies (ii-ok n)
                (and (ok n) (ii-ok (1+ n))))))
```

$\lfloor$ Approaches - Model Checking $\rfloor$

- Explore an "effective" finite state graph of a system searching for failures

  – Specification is usually provided by a temporal logic formula: e.g. an invariant in CTL would be $AG(\texttt{ok})$

  – System definition languages: Verilog HDL, VHDL, SMV, Murϕ, SPIN, Limited variants of C/C++, etc.

  – Model checkers are generally classified into explicit-state and implicit-state

  – Several algorithms exist to reduce large-state systems to effectively finite *abstract* state systems: symmetry reductions, partial order reductions, etc.

- Hybrid approaches: too many to enumerate, but most involve some form of abstraction.

⌊ **Our Approach - Phase 1** ⌋

- Assume the definition of a term rewrite function `rewrt` which takes a term as an input and produces the rewritten term

- For a predicate ϕ, denote ϕ' as the term:
`(rewrt `((lambda (n) ,`$\phi$`) (1+ n)))`

- Assume the following function definition:

```
(defun state-ps (trm)
  (cond ((or (atom trm) (quotep trm)) ())
        ((eq (first trm) 'if)
         (union-equal (state-ps (second trm))
           (union-equal (state-ps (third trm))
                        (state-ps (fourth trm)))))
        (t (and (state-predp trm) (list trm)))))
```

- Compute the least set of predicates Ψ s. t. :
(a) the target invariant predicate $\tau \in \Psi$, and
(b) for every $\phi \in \Psi$, `(state-ps` ϕ'`)` $\subseteq \Psi$

⌊ Our Approach - Phase 2 ⌋

- Given the finite predicate set Ψ, we first compute the finite set of input predicates Γ

  – For each predicate ϕ in Ψ and Γ, define a boolean variable $bv(\phi)$

  – The boolean variables for Ψ are state var.s and the variables for Γ are input var.s

- For each α in Ψ, we replace the predicate subterms ϕ in α' with $bv(\phi)$

  – This gives us a propositional next-state function for $bv(\alpha)$ in terms of the state and input boolean var.s

- Explore the graph of nodes defined by next-state functions starting from initial node

  – If no path is found to a node where $bv(\tau)$ is `nil`, then return Q.E.D.

  – Otherwise, return a pruned version of the failing path to the user for further analysis

⌊ Our Approach - Elaborations ⌋

- The function `(state-predp trm)` is essentially defined as:

```
(defun state-predp (trm)
  (and (not (intersectp-eq (all-fnnames trm) '(t+ hide)))
       (equal (all-vars trm) '(n))))
```

  – Thus, the user can cause introduce an input predicate by introducing a `hide`

- We chose to define our own term rewriter because simplicity is more important than efficiency

  – The rewriter does extract rewrite rules from the current ACL2 world

- Our "model checker" is an compiled, optimized (to an extent), explicit-state, breadth-first search through the predicate state graph

- The prover also supports assume-guarantee reasoning through the use of `forced` hypothesis

⌊ Mutual Exclusion Continued ⌋

• Beginning with `(ok n)`, the prover generates the following set of predicates:

```
(ok n)
(equal (status (a) n) ':critical)
(equal (status (b) n) ':critical)
(equal (status (a) n) ':try)
(equal (status (b) n) ':try)
(in-critical n)
(equal (critical-id n) (a))
(equal (critical-id n) (b))
```

• The resulting graph has 20 nodes and verifies that `(ok n)` is never `nil`

• We can further reduce the graph to 6 nodes by hiding `:try` terms:

```
(defthm coerce-try-status-to-input
  (equal (equal (status p n) ':try)
         (hide (equal (status p n) ':try))))
```

⌊ MESI cache example-1 ⌋

- More complex example: a high-level definition of the MESI cache coherence protocol

  – Ok, technically we only model ESI cache states

- System defined by following state variables:

  – `(mem c n)` – shared memory data for cache-line `c`

  – `(cache p c n)` – data for cache-line `c` at proc. `p`

  – `(valid c n)` and `(excl c n)` – sets of processor id.s which define the ESI cache states

- We will need a few constrained functions:

```
(encapsulate (((proc *) => *) ((op *) => *)
              ((addr *) => *) ((data *) => *))
 (local (defun proc (n) n)) (local (defun op (n) n))
 (local (defun addr (n) n)) (local (defun data (n) n)))

(encapsulate (((c-l *) => *)) (local (defun c-l (a) a)))
```

```
(define-system mesi-cache
 (mem (c n) nil
   (cond ((/= (c-l (addr n)) c) (mem c n-))
         ((and (= (op n) :flush)
               (in1 (proc n) (excl c n-)))
          (cache (proc n) c n-))
         (t (mem c n-))))

 (cache (p c n) nil
   (cond ((/= (c-l (addr n)) c) (cache p c n-))
         ((/= (proc n) p) (cache p c n-))
         ((or (and (= (op n) :fill) (not (excl c n-)))
              (and (= (op n) :fille) (not (valid c n-))))
          (mem c n-))
         ((and (= (op n) :store) (in1 p (excl c n-)))
          (s (addr n) (data n) (cache p c n-)))
         (t (cache p c n-))))

 (excl (c n) nil
   (cond ((/= (c-l (addr n)) c) (excl c n-))
         ((and (= (op n) :flush)
               (implies (excl c n-)
                        (in1 (proc n) (excl c n-))))
          (sdrop (proc n) (excl c n-)))
         ((and (= (op n) :fille) (not (valid c n-)))
          (sadd (proc n) (excl c n-)))
         (t (excl c n-))))

 (valid (c n) nil
   (cond ((/= (c-l (addr n)) c) (valid c n-))
         ((and (= (op n) :flush)
               (implies (excl c n-)
                        (in1 (proc n) (excl c n-))))
          (sdrop (proc n) (valid c n-)))
         ((or (and (= (op n) :fill) (not (excl c n-)))
              (and (= (op n) :fille) (not (valid c n-))))
          (sadd (proc n) (valid c n-)))
         (t (valid c n-)))))
```

⌊ MESI cache example-3 ⌋

• Property: the value read by a processor is the last value stored.

• A codification in ACL2 of this property as the target invariant `(ok n)`:

```
(encapsulate (((p) => *) ((a) => *))
  (local (defun p () t)) (local (defun a () t)))

(define-system mesi-specification
 (a-dat (n) nil
   (if (and (= (addr n) (a))
            (= (op n) :store)
            (in1 (proc n) (excl (c-l (a)) n-)))
       (list (data n))
     (a-dat n-)))

 (ok (n) t
   (if (and (a-dat n-)
            (= (proc n) (p))
            (= (addr n) (a))
            (= (op n) :load)
            (in (p) (valid (c-l (a)) n-)))
       (= (g (a) (cache (p) (c-l (a)) n-))
          (car (a-dat n-)))
     (ok n-))))
```

⌊ MESI cache example-4 ⌋

- Key rewrite rule to introduce case splits on the exclusive set `(excl c n)`:

```
(defthm in1-force-split
  (equal (in1 e s)
         (cond ((not s) nil)
               ((c1 s) (equal e (scar s)))
               (t (hide (in1 e s))))))
```

- Prover generates following predicate set and explores resulting graph (48 nodes):

```
  (ok n)
  (a-dat n)
  (valid (c-l (a)) n)
  (in (p) (valid (c-l (a)) n))
  (excl (c-l (a)) n)
  (in (p) (excl (c-l (a)) n))
  (c1 (excl (c-l (a)) n))
  (equal (scar (excl (c-l (a)) n)) (p))
  (equal (g (a) (cache (scar (excl (c-l (a)) n))
                       (c-l (a)) n))
         (car (a-dat n)))
  (equal (g (a) (cache (p) (c-l (a)) n))
         (car (a-dat n)))
  (equal (g (a) (mem (c-l (a)) n))
         (car (a-dat n)))
```

⌊ Conclusions and Future Work ⌋

- Prover can be effective but requires thought:
  - Careful consideration of system definition and specification relative to existing operators and rewrite rules
  - Determination of which terms should be hidden and the possible addition of auxiliary variables
- Improvements to the Prover:
  - Interfaces to external model checkers for Phase 2
  - Compress/Reduce resulting predicate graph based on equality reasoning between state and input predicates
  - Various improvements to built-in “model checker”
- Many more example systems and effort to integrate with RTL definitions and existing library
- Need to develop more comprehensive compositional methodology