Reducing Invariant Proofs to Finite Search via Rewriting

ACL2 Workshop 2004

Austin, Texas, November 18, 2004

Rob Sumners and Sandip Ray

robert.sumners@amd.com,
sandip@cs.utexas.edu
What are Invariants?

- A Term is either a variable symbol, a quoted constant, or a function application

  - Example:
    \[(\text{cons} \ (\text{binary--} \ x \ (\text{quote} \ 1)) \ \text{'(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

• 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 examples of this phenomenon from the ACL2 community and other formal methods communities
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 \texttt{critical} state at the same time

- Codified as the invariant (ok n):

$(\text{encapsulate } ((a) \Rightarrow *) ((b) \Rightarrow *))$
$(\text{local } (\text{defun } a () 1))$
$(\text{local } (\text{defun } b () 2))$
$(\text{defthm } a-=/=b \ (\text{not } (\text{equal } (a) (b)))))$

$(\text{defun } ok (n))$
$(\text{not } (\text{and } (= (\text{status } (a) n) \text{critical})$
$(= (\text{status } (b) n) \text{critical}))))$
• 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:

  \[
  \begin{align*}
  (\text{defun} & \ ii-ok-for1 \ (n \ i) \\
  (\text{iff} & \ (= \ (\text{status} \ i \ n) :\text{critical}) \\
  (\text{and} & \ (= \ (\text{critical-id} \ n) \ i))))
  \end{align*}
  \]

  \[
  \begin{align*}
  (\text{defun} & \ ii-ok \ (n) \\
  (\text{and} & \ (\text{ii-ok-for1} \ n \ (a)) \ (\text{ii-ok-for1} \ n \ (b))))
  \end{align*}
  \]

  \[
  \begin{align*}
  (\text{defthm} & \ ii-ok-is-inductive-invariant \\
  (\text{and} & \ (\text{ii-ok} \ (t0)) \\
  (\text{implies} & \ (\text{ii-ok} \ n) \\
  (\text{and} & \ (\text{ok} \ n) \ (\text{ii-ok} \ (t+ \ n)))))
  \end{align*}
  \]

  \[
  (\text{defthm} \ ok-is-invariant \ (\text{ok} \ n))
  \]
• 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 $AG(\text{ok})$

  – System definition languages: Verilog HDL, VHDL, SMV, Mur$\phi$, 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 \texttt{rewrt} which takes a term as an input and produces the rewritten term.

- For a predicate \( \phi \), denote \( \phi' \) as the term:
  \[
  \texttt{(rewrt '((lambda (n) ,\phi) (t+ n)))}
  \]

- Assume the following function definition:
  \[
  \text{(defun state-ps (trm)}
  (\text{cond}}
  \begin{align*}
  & (\text{"(or (atom trm) (quotep trm)) ()")} \\
  & (\text{"(eq (first trm) 'if)")} \\
  & (\text{"(union-equal (state-ps (second trm))")} \\
  & (\text{"(union-equal (state-ps (third trm))")} \\
  & (\text{"(state-ps (fourth trm)))")})
  \\
  & (t (\text{"(and (state-predp trm) (list trm)))")})
  \end{align*}
  \]

- Compute the least set of predicates \( \Psi \) s. t.: 
  \( a \) the target invariant predicate \( \tau \in \Psi \), and 
  \( b \) for every \( \phi \in \Psi \), \( \text{(state-ps \phi')} \subseteq \Psi \).
• From the $\phi'$, we compute a finite set of input (non-state) predicates $\Gamma$

  — For each predicate $\alpha$ in $\Psi \cup \Gamma$, define a boolean variable $bv(\alpha)$

• For each $\phi$ in $\Psi$, we replace the predicate sub-terms $\alpha$ in $\phi'$ with $bv(\alpha)$

  — This gives us a next-value function for computing the next value of $bv(\phi)$ in terms of the current values of the boolean variables

• Explore the abstract graph defined by the next-value functions for $bv(\Psi)$

  — nodes in the graph are valuations of the variables $bv(\Psi)$ and an edge exists from one node to the next if a valuation of $bv(\Gamma)$ exists
  — 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 introduce an input predicate by introducing a `hide`

- We chose to define our own term rewriter for numerous reasons

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

- Our “model checker” is a compiled, optimized (to an extent), explicit-state, breadth-first search through the abstract graph

- The prover also supports assume-guarantee reasoning through the use of `forced` hypothesis
• Beginning with $\tau = (\text{ok } n)$, the prover generates the following set of predicates $\Psi$:

\[
\begin{align*}
(\text{ok } n) \\
(\text{equal } (\text{status } (a) \ n) \ '':\text{critical}) \\
(\text{equal } (\text{status } (b) \ n) \ '':\text{critical}) \\
(\text{equal } (\text{status } (a) \ n) \ '':\text{try}) \\
(\text{equal } (\text{status } (b) \ n) \ '':\text{try}) \\
(\text{in-critical } n) \\
(\text{equal } (\text{critical-id } n) \ (a)) \\
(\text{equal } (\text{critical-id } n) \ (b))
\end{align*}
\]

• The resulting abstract graph has 20 nodes and verifies that $(\text{ok } n)$ is never nil.

• We can further reduce the graph to 6 nodes by hiding $:\text{try}$ terms:

\[
\begin{align*}
(\text{defthm } \text{coerce-try-status-to-input} \\
(\text{equal } (\text{equal } (\text{status } p \ n) \ '':\text{try}) \\
(\text{hide } (\text{equal } (\text{status } p \ n) \ '':\text{try})))
\end{align*}
\]
• Another example: a high-level definition of the ESI cache coherence protocol

• 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:

\[
\text{(encapsulate } (((\text{proc } *)) => *)) ((\text{op } *)) => *) \\
\quad ((\text{addr } *)) => *) ((\text{data } *)) => *)
\]

\[
\text{(local (defun proc (n) n)) (local (defun op (n) n))} \\
\quad \text{(local (defun addr (n) n)) (local (defun data (n) n))}
\]

\[
\text{(encapsulate } (((\text{c-l } *)) => *)) \text{(local (defun c-l (a) a)))}
\]
(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-))))
(t (valid c n-))))

13
• Property: the value read by a processor is the last value stored.

• A codification in ACL2 of this property as the target invariant \((\text{ok } n)\):

\[
\text{(encapsulate } (((p) \Rightarrow *) ((a) \Rightarrow *)) \\
\text{ (local (defun p () t)) (local (defun a () t)))}
\]

\[
\text{(define-system mesi-specification} \\
\text{ (a-dat (n) nil} \\
\text{ (if (and (= (addr n) (a))} \\
\text{ (= (op n) :store) } \\
\text{ (in1 (proc n) (excl (c-l (a)) n-)))} \\
\text{ (data n) } \\
\text{ (a-dat n-))))}
\]

\[
\text{(ok (n) t} \\
\text{ (if (and (= (proc n) (p))} \\
\text{ (= (addr n) (a))} \\
\text{ (= (op n) :load) } \\
\text{ (in (p) (valid (c-l (a)) n-)))} \\
\text{ (= (g (a) (cache (p) (c-l (a)) n-)) (a-dat n-))} \\
\text{ (ok n-))))}
\]
• Key rewrite rule to introduce case splits on the exclusive set \( (\text{excl } c \ n) \):

\[
\text{(defthm in1-case-split} \\
\quad \text{(equal } \text{(in1 } e \ s) \\
\qquad \text{(cond (not } s) \text{ nil) \\
\qquad \quad ((c1 } s) \text{ (equal } e \text{ (scar } s)) \\
\qquad \quad (t \text{ (hide } \text{(in1 } e \ s)))))
\]

• Prover generates following predicate set and explores resulting graph (11 nodes):

\[
\text{(ok } n) \\
\text{(valid } (\text{c-l } (a)) \ n) \\
\text{(in } (p) \text{ (valid } (\text{c-l } (a)) \ n)) \\
\text{(excl } (\text{c-l } (a)) \ n) \\
\text{(c1 } (\text{excl } (\text{c-l } (a)) \ n)) \\
\text{(equal } (\text{scar } \text{(excl } (\text{c-l } (a)) \ n)) \ (p)) \\
\text{(equal } (\text{a-dat } n) \ (g \ (a) \ (\text{mem } (\text{c-l } (a)) \ n))) \\
\text{(equal } (\text{a-dat } n) \ (g \ (a) \ (\text{cache } (p) \ (\text{c-l } (a)) \ n))) \\
\text{(equal } (\text{a-dat } n) \ (g \ (a) \ (\text{cache } (\text{scar } \text{(excl } (\text{c-l } (a)) \ n)) \ (\text{c-l } (a)) \ n))))
\]
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
  – Better methodology for prover use and user feedback

• Many more example systems and effort to integrate with RTL definitions and existing library

• Need to develop more comprehensive compositional methodology