Formal Verification of LabVIEW Programs with ACL2: Progress Report on Handling State

Matt Kaufmann

April, 2008
OUTLINE

 BACKGROUND

 ▶ The problem of state
 ▶ Hierarchy (work in progress)
OUTLINE

- Background
  - Brief history
  - ACL2 representation
  - Main verification idea
- The problem of state
- Hierarchy (work in progress)
Jeff Kodosky started playing around in 2004 with the idea of verifying a LabVIEW program.
Warren Hunt and J Moore met on occasion with Jeff and Jacob Kornerup over several years, culminating with NI engaging Grant Passmore as an intern in 2005.
Grant developed an approach to prove Gauss’s theorem that the sum of the integers from 1 to n is n*(n+1)/2.
Summer 2007: Matt Kaufmann developed an alternate approach to model LabVIEW programs, including loop structures, directly as ACL2 functions. Grant updated the infrastructure accordingly.
Fall 2007: Grant transferred infrastructure support to Mark Reitblatt, NI intern from UT CS. Mark has worked with Matt on further automating the loop verification.
Since then: Matt has been working on extending the previous work to handle LabVIEW diagrams with state. Also, Mark has looked into applying model checking.
Every module, primitive or not, takes and returns a single alist that we call a *record*, by calling $S*$, “set”.

Every wire returns a LabVIEW data value, obtained by applying $G$, “get”, to a record.

```
|----------------------------------- n
| |
|---N0 |----N1 |
|x --|-- X0 --| | | | |
| | + |-- W0 --| 1+ |-- Z0 --|-- z |
| | |---| |----| |
|y --|-- Y0 --| | | | |
| |---| |----| |
```
(DEFUN X0 (IN) (G :X IN))
(DEFUN Y0 (IN) (G :Y IN))
(DEFUN N0 (IN) (S* :T0 (+ (X0 IN) (Y0 IN))))
(DEFUN W0 (IN) (G :T0 (N0 IN)))
(DEFUN N1 (IN) (S* :T0 (1+ (W0 IN))))
(DEFUN Z0 (IN) (G :T0 (N1 IN)))
MAIN VERIFICATION IDEA

- An assertion is simply a Boolean-valued wire that can be checked at runtime.

- Goal: prove that each assertion is true

- Earlier focus: For-loops and while-loops
OUTLINE

▶ Background
▶ **The problem of state:**
  ▶ **Producer-consumer scenario**
  ▶ **Valid traces**
    ▶ A simple producer-consumer example using global variables
    ▶ A simple producer-consumer example using a feedback loop
  ▶ Atomic read-modify-write using sub-VIs
▶ Hierarchy (work in progress)
Producer-consumer scenario (p. 1)
Producer-consumer scenario (p. 2)

N1: Wr B, 1  
N2: Rd B  
N3: Wr A, 1  
N4: Rd A  
N5: Wr A, 2  
N6: Wr B, 2

We want to prove the following.

- N4 reads 1 or 2 for A.
- If N2 reads 2 for B, then N4 reads 1 for A.
Valid traces (p. 1)

For “If N2 reads 2 for B, then N4 reads 1 for A”: This theorem will be stated as a property of all valid computation traces — node sequences.

N1: Wr B, 1
^ N5: Wr A, 2
^ N2: Rd B >? N6: Wr B, 2
^ N3: Wr A, 1
^ N4: Rd A
Valid traces (p. 1)

For “If N2 reads 2 for B, then N4 reads 1 for A”: This theorem will be stated as a property of all valid computation *traces* — node sequences.

\[
\begin{align*}
\text{N1: Wr B, 1} & \quad \text{N5: Wr A, 2} \\
\quad & \quad \\
\text{N2: Rd B} & \quad \text{N6: Wr B, 2} \\
\quad & \\
\text{N3: Wr A, 1} & \\
\quad & \\
\text{N4: Rd A} & \\
\end{align*}
\]

We specify node pairs (N . N’) such that N must fire before N’:

\[
\text{(valid-tracep-setup st1)}
\]

\[
\begin{align*}
\text{(n1 . n2)} & \\
(n2 . n3) & \\
(n3 . n4) & \\
(n5 . n6)) & \\
\end{align*}
\]
Valid traces (p. 2)

Generated by above valid-tracep-setup call:

(DEFUN ST1$VALID-TRACEP (LST)
  (AND (NO-DUPLICATESP-EQUAL LST)
       (PREC-LST (ST1$PREC-REL) LST)))

Examples:

ACL2  !>(st1$valid-tracep (reverse '(n1 n2 n3 n4 n5 n6)))
   T
ACL2  !>(st1$valid-tracep (reverse '(n1 n5 n2 n3 n6 n4)))
   T
ACL2  !>(st1$valid-tracep (reverse '(n1 n5 n2)))
   T
ACL2  !>(st1$valid-tracep (reverse '(n4 n5 n6)))
   NIL
ACL2  !>(st1$valid-tracep (reverse '(n2 n1)))
   NIL
ACL2  !>
Consider:

:trans1 (valid-tracep-setup st1
    ((n1 . n2)
    (n2 . n3)
    (n3 . n4)
    (n5 . n6)))

Here we edit away some output. Note that some hints use functional instantiation.
Valid traces (p. 4)

(PROGN (DEFUN ST1$PREC-REL ()
  '((N1 . N2)
    (N2 . N3)
    (N3 . N4)
    (N5 . N6)))
(DEFUN ST1$VALID-TRACEP (LST)
  (AND (NO-DUPLICATESP-EQUAL LST)
       (PREC-LST (ST1$PREC-REL) LST)))
(DEFTHM ST1$VALID-TRACEP-FORWARD-TO-NO-DUPLICATESP-EQUAL
  (IMPLIES (ST1$VALID-TRACEP TRACE)
            (NO-DUPLICATESP-EQUAL TRACE))
  :RULE-CLASSES :FORWARD-CHAINING)
(DEFTHM ST1$VALID-TRACEP-FORWARD-TO-PREC-N1-N2
  (IMPLIES (AND (ST1$VALID-TRACEP TRACE)
                (MEMBER-EQUAL 'N2 TRACE))
            (MEMBER-EQUAL 'N1
                         (MEMBER-EQUAL 'N2 TRACE)))
  :RULE-CLASSES :FORWARD-CHAINING)
... ; similarly for N2-N3, N3-N4, N5-N6
(DEFTHM ST1$VALID-TRACEP-FORWARD-TO-PREC-N1-N2$2
  (IMPLIES (AND (ST1$VALID-TRACEP TRACE)
                (EQUAL 'N2 (CAR TRACE)))
            (MEMBER-EQUAL 'N1 (CDR TRACE)))
  :RULE-CLASSES :FORWARD-CHAINING)
... ; similarly for N2-N3$2, N3-N4$2, N5-N6$2
(IN-THEORY (DISABLE ST1$VALID-TRACEP))
(DEFTHM ST1$VALID-TRACEP-CDR
  (IMPLIES (ST1$VALID-TRACEP TRACE)
            (ST1$VALID-TRACEP (CDR TRACE)))
(DEFTHM ST1$VALID-TRACEP-MEMBER-EQUAL
  (IMPLIES (ST1$VALID-TRACEP TRACE)
            (ST1$VALID-TRACEP (MEMBER-EQUAL NODE TRACE)))
(DEFCONST *ST1$NODES* '(N1 N5 N6 N2 N3 N4))
OUTLINE

- Background
- **The problem of state:**
  - Producer-consumer scenario
  - Valid traces
  - **Producer-consumer with global variables**
  - Producer-consumer with a feedback loop
  - Atomic read-modify-write using sub-VIs
- Hierarchy (work in progress)
We’re skipping most technical detail (time limitations).

See .lisp files (certified books) for details, including some interesting technical challenges. I’m happy to serve as tour guide.
We return to the example already presented:

N1: Wr B, 1  N5: Wr A, 2
N2: Rd B  N6: Wr B, 2
N3: Wr A, 1
N4: Rd A

We want to prove the following.

- N4 reads 1 or 2 for A.
- If N2 reads 2 for B, then N4 reads 1 for A.

The next slides illustrate our translation.
Basic node and wire functions are based on the state at the time the node or wire gets its value, e.g.:

; Node N3 does a write, so returns nothing:
(defun n3@ (in st)
  nil)

; Node N4 returns a record with the value of A.
(defun n4@ (in st)
  (s* :t0 (g :a st)))

; This wire (for terminal T0 of N4) is the value that has been read for A.
(defun n4-t0@ (in st)
  (g :t0 (n4@ in st)))
Producer-consumer with global variables (p. 3)

Next, we say how state is updated.

(defun st1$state-step (node in st)
  (case node
    (n1 (s :b 1 st)) ; Wr B, 1
    (n3 (s :a 1 st)) ; Wr A, 1
    (n5 (s :a 2 st)) ; Wr A, 2
    (n6 (s :b 2 st)) ; Wr B, 2
    (otherwise st)))

(defun st1$state-rec (in st trace)
  (if (consp trace)
      (st1$state-step (car trace) in
          (st1$state-rec in st (cdr trace)))
      st))

(defun st1$state (node in st trace)
  (st1$state-rec in st (cdr (member-equal node trace))))
Then, we define the actual node and wire functions in terms of the state as of the time the diagram is first entered, e.g.:

; Node N3 does a write, so returns nothing:
(defun n3 (in st trace)
  (n3@ in (st1$state 'n3 in st trace)))

; Node N4 returns a record with the value it reads.
(defun n4 (in st trace)
  (n4@ in (st1$state 'n4 in st trace)))

; This wire (for terminal T0 of N4) is the value that has been read.
(defun n4-t0 (in st trace)
  (n4-t0@ in (st1$state 'n4 in st trace)))
Example that evaluates to T:

(let ((trace (reverse '(n1 n2 n3 n4 n5 n6)))))
  (and (st1$valid-tracep trace)
       (equal (n4 nil '((:a . 0) (:b . 0)) trace)
              '((:T0 . 1)))))

N1: Wr B, 1
N2: Rd B
N3: Wr A, 1
N4: Rd A
N5: Wr A, 2
N6: Wr B, 2
**Producer-consumer with global variables (p. 6)**

**First Theorem: N4 reads 1 or 2.** The following invariant could be automatically generated. Proof using functional instantiation replaces explicit induction by a base and an induction step.

```
(defun st1$state-inv1 (in st trace)
  (implies (member-equal 'n3 trace)
    (member-equal (g :a (st1$state-rec
      in st trace))
      '(1 2))))
```

The key observation is that N4 is after N3:

N3: Wr A, 1
N4: Rd A

The invariant then yields the theorem:

```
(implies (and (st1$valid-tracep trace)
  (subsetp-equal *st1$nodes* trace))
  (member-equal (n4-t0 in st trace) '(1 2)))
```
Second Theorem: If N2 reads 2 for B, then N4 reads 1 for A.

\[(\text{implies} \ (\text{and} \ (\text{st1$valid-tracep} \ \text{trace})
\qquad \text{(subsetp-equal} *\text{st1$nodes}* \ \text{trace})
\qquad \text{(equal} \ (\text{n2-t0} \ \text{in} \ \text{st} \ \text{trace}) \ 2))
\qquad \text{(equal} \ (\text{n4-t0} \ \text{in} \ \text{st} \ \text{trace}) \ 1))\]
Producer-consumer with global variables (p. 8)

N1: Wr B, 1       N5: Wr A, 2
N3: Wr A, 1
N4: Rd A [1?]

Our reasoning goes as follows.

N6 « N2 {by invariant:}
   If N1 « N and not N6 « N, then value of B is 1
N4 reads 1 for A {by invariant:}
   If N5 « N3 « N, the value of A at N is 1

The user is expected to create the two invariants, but our .lisp file suggests that the system could then prove them automatically.
The above example is file `state-1.lisp`. We have created two elaborations:

- **state-2.lisp**
  Re-working of `state-1.lisp`, reading directly from inputs instead of using constants.

- **state-3.lisp**
  Re-working of `state-2.lisp`, but using proper wires for inputs and thus using mutual-recursion for wire, node, and state functions.
OUTLINE

- Background
- The problem of state:
  - Producer-consumer scenario
  - Valid traces
  - Producer-consumer with global variables
  - Producer-consumer with a feedback loop
  - Atomic read-modify-write using sub-VIs
- Hierarchy (work in progress)
Producer-consumer with a feedback loop (p. 1)

The key element for this version of the problem is a latch VI, which is *non-reentrant*: only one instance is being evaluated at a time.

File `state-4.lisp` includes this “wonderful” graphic:

```
; -----           
; -- wrp --      |
; -- din --  ITE | -- out --
; st --  V
; / ^   -----    
; 0     |
;        __________________________
```

The main VI instantiates two different (but isomorphic) such latches, A and B, each three times (two writes and one read, each):
Producer-consumer with a feedback loop (p. 2)
Producer-consumer with a feedback loop (p. 3)

This example is similar to the earlier one, though more complex. We see a first attempt to handle hierarchical state elements.

```lisp
(defun st1$state-step (node in new-st st trace)
  (declare (xargs :measure (st1$measure node trace :state-step)))
  (if (and (st1$valid-tracep trace)
           (member-equal node trace))
    (case node
      (n1 (s :latch-b ; WR B, bl
           (latch-b{post-state} (s* :wrp (n1-wrp-t0 in st trace)
                                   :din (in-b1 in st trace)
                                   (g :latch-b new-st))
           new-st))
      (n2 (s :latch-b ; Rd B
           (latch-b{post-state} (s* :wrp (n2-wrp-t0 in st trace)
                                   :din 0)
           (g :latch-b new-st))
           new-st))
    ....
    (otherwise new-st))
  new-st))
```
Producer-consumer with a feedback loop (p. 4)

The proofs of the two theorems are similar to the earlier ones. But there are no “@” functions – state as of entry to a node doesn’t tell you the state at input wires. (Initially I got this wrong!)

However, we first need to prove invariants about the bits of state indicating whether the feedback element has ever been entered, e.g. for Latch A:

(implies
  (st1$valid-tracep trace)
  (let ((n3p (member-equal 'n3 trace))
       (n5p (member-equal 'n5 trace)))
   (equal (g :st{first}
      (g :latch-a
        (st1$state-rec in st trace)))
    (if (or n3p n5p)
      nil
      (g :st{first} (g :latch-a st))))))
OUTLINE

- Background
- The problem of state:
  - Producer-consumer scenario
  - Valid traces
  - Producer-consumer with global variables
  - Producer-consumer with a feedback loop
  - Atomic read-modify-write using sub-VIs
- Hierarchy (work in progress)
Atomic read-modify-write using sub-VIs (p. 1)

Sub-VI Inc, state-5.lisp (to be fixed like state-4.lisp):

; ----
; en -- | |
; st0 -- | 1+ | -- out --
; /\ | | V
; / | ---- |
; 0 | ___________________
; ________________

Main VI from state-5.lisp: two writes, then a read.

; N1: Inc[En=T]   N2: Inc[En=T]
; ________________________________
; N3: Inc[En=NIL]

Theorem proved:

(implies (and (vi0$valid-tracep trace)
                (g :st0{first} (g :inc st))
                (subsetp-equal *vi0$nodes* trace))
         (equal (n3-out in st trace) 2))
OUTLINE

- Background
- The problem of state:
  - Producer-consumer scenario
  - Valid traces
  - Producer-consumer with global variables
  - Producer-consumer with a feedback loop
  - Atomic read-modify-write using sub-VIs
- Hierarchy (work in progress)
We do not yet handle loops with state. Work in progress:

- A rather detailed 6-page plan that could deal nicely with loops and other hierarchy.
- Main idea: notion of node is extended to hierarchical node: in essence, a path of enclosing node instances down towards a leaf node.
- A trace is then a list of hierarchical nodes. A valid trace must respect loop indices, in particular.
Conclusion (p. 1)

There’s a good start on handling state:

- Trace model, with helpful (and proved) supporting rules
- Examples have been worked
- Hierarchy has been considered

The next step is to implement the hierarchical approach to work the motivating example from Jacob Kornerup:

There are two loops with 100 iterations each, one incrementing and the other decrementing a global integer at each iteration. The increment/decrement operations are atomic. Prove that the final value of the global equals its initial value.
Guiding principles to balance are the following.

- Work simple examples to develop methodologies.
- But keep in mind future automation and scalability.
- And use a suitable translation:
  - Stick to the earlier, simpler approach if there is no state.
  - Sub-VIs using feedback loop don’t need interpreter, since state isn’t updated until sub-VI is exited.