Formal Verification of LabVIEW Diagrams

<table>
<thead>
<tr>
<th>Matt Kaufmann</th>
<th>Jacob Kornerup</th>
<th>Mark Reitblatt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dept. of Computer Sciences, University of Texas</td>
<td>National Instruments, Inc.</td>
<td>Dept. of Computer Sciences, University of Texas National Instruments, Inc.</td>
</tr>
</tbody>
</table>
Outline

• Project History
• LabVIEW Overview
• Overview of approach
• Walk through example verification
• Conclusion
Project History

• 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 as an intern in 2005

• Summer 2007: Alternate approach models LabVIEW programs, including loop structures, directly as ACL2 functions. At the end of the summer Grant left for Edinburgh and transferred his work to Mark Reitblatt

• Current: Approach has been fully automated, expanded and used to verify a dozen examples
LabVIEW (in brief)

- Graphical dataflow language (G) with control structures
- Shift register memory elements
- Separate Front (user interface) and Back (implementation) panels
Why LabVIEW?

- Mostly functional
- Memory safe
- Simple control structures
Our Approach

- Add “assertion” blocks to LabVIEW/G
Our Approach (cont.)

- Translate LabVIEW/G diagrams into ACL2 functions (shallow embedding)
- Each node takes a record (IN) as input
- Returns a record binding its outputs to terminal names
- Wires extract values from records
Naming

- LabVIEW/G doesn’t allow naming of (most) nodes
- Human readability is essential to understanding proofs
- Auto-naming of nodes based on type
Naming (cont.)

- Fn nodes are named as fn_type-number
  - ADD-1
- Constant nodes are named by value
  - CONSTANT[0]-2
- Third instance of the constant ‘0’
Naming (cont.)

- Wires are named a little differently
- Each wire retrieves one terminal from one node
- Wire named after its source

CONSTANT[0]-2<\_T\_0>
• (G :key rec) returns the value associated with :key in rec

• (S* :key1 val1 :key2 val2 ...) creates new record binding :keyi to vali
Our Approach (cont.)

- Translate assertions into proof obligations

(DEFTHM ASSERTION-BLOCK-HOLDS
  (IMPLIES (AND (NATP (G :NUMBER IN)))
    (G :ASN (ASSERTION-BLOCK IN))))
Limitations

- Currently only for-loops are automated
- We use unbounded arithmetic, so this is a theorem for us, but not for LabVIEW/G
Loop Assertions

• Assertions about loops (in general) require inductive proofs

• We split loop assertions into “top” assertions and loop invariants
Loop Assertions (cont.)
Loop Invariant
Loop Assertion
Proving Loop Assertions

- Hold the user’s hand to prove invariants
- Autogenerate highly structured proof scaffolding
- Strictly guide proof process through theory control
LabVIEW Loops

• We separate for-loop structures into 4 ACL2 functions

• $step$ function

• Executes loop body and binds outputs to next iteration inputs

(DEFUN FOR-LOOP$STEP (IN)
 (S : |_T_4| (G : |_T_1| ( |_N_5| IN)) IN))
LabVIEW Loops (cont.)

- $loop function
- Compares loop counter to loop bound
- Updates loop counter and calls $step fn

```
(DEFUN FOR-LOOP$LOOP (N IN)
 (DECLARE (XARGS :MEASURE (NFIX (- N (G :LC IN))))
 (COND ((OR (>= (G :LC IN) N)
               (NOT (NATP N))
               (NOT (NATP (G :LC IN))))
          IN)
       (T (FOR-LOOP$LOOP N (S :LC (1+ (G :LC IN))
                               (FOR-LOOP$STEP IN))))))
```
LabVIEW Loops (cont.)

- $init$ function
  - Binds loop variables to initial values

```lisp
(DEFUN FOR-LOOP$LOOP$INIT (IN)
  (S* :LC 0
    :_T_2 (CONSTANT[10]-1<_T_0> IN)
    :_T_4 (CONSTANT[0]-0<_T_0> IN)))
```
LabVIEW Loops (cont.)

- Top function
  - Binds loop bound and calls $\text{loop fn}$ with results of $\text{init fn}$

\[
\text{(DEFUN-N FOR-LOOP (IN)}
\text{(FOR-LOOP-SRN$\text{LOOP}$ (CONSTANT[10]-1$_$T_0$>$ IN)
\text{(FOR-LOOP-SRN$\text{LOOP}$\text{INIT IN))))}
\]
LabVIEW Structures

- LabVIEW loops are split into inner and outer structures
  - Inner structures are called “Self-reference Nodes” (SRN)
  - SRN nodes contain the body of the loop
  - Outer nodes map external values to internal names
Generic Theory

• We use a generic theory to avoid induction in the invariant proof

• Use encapsulate to define a generic $step, $loop and $prop (invariant)

• Prove that if $prop holds on entry to $loop and is preserved by $step then it holds when $loop is run
Extend Loop Invariant

\[
(\text{DEFUN } |\text{LOOP-INV-SRN$PROP}| \ (N \ \text{IN}) \\
(\text{DECLARE} \ (\text{IGNORABLE} \ N)) \\
(\text{AND} \ (|\text{LOOP-INV-SRN$HYPS}| \ \text{IN}) \\
(\text{EQUAL} \ N \ (G : |_T_3| \ \text{IN})) \\
(G : \text{ASN} \ (\text{ACL2-LOOP-INV} \ \text{IN}))))
\]

- LOOP-INV-SRN$HYPS is a type predicate that recognizes the types on the inputs to LOOP-INV-SRN
- ACL2-LOOP-INV is the name of the loop invariant
Loop Inv. is Preserved

(DEFTHMDL LOOP-INV-SRN$PROP{FOR-LOOP-SRN$STEP}|
 (IMPLIES (AND (NATP (G :LC IN))
             (< (G :LC IN) N)
             (|LOOP-INV-SRN$PROP| N IN))
             (|LOOP-INV-SRN$PROP| N
             (S :LC (1+ (G :LC IN))
             (|FOR-LOOP-SRN$STEP| IN))))))

- Note that this lemma is disabled
Use Generic Theory

(DETHML |LOOP-INV-SRN$PROP{FOR-LOOP-SRN}|)
(IMPLIES (AND (NATP N)
  (NATP (G :LC IN))
  (|LOOP-INV-SRN$PROP| N IN))
  (|LOOP-INV-SRN$PROP| N (| FOR-LOOP-SRN$LOOP| N IN)))

:HINTS
  ("Goal" :BY (:FUNCTIONAL-INSTANCE
    LOOP-GENERIC-THM
    (STEP-GENERIC |FOR-LOOP-SRN$STEP|)
    (PROP-GENERIC |LOOP-INV-SRN$PROP|)
    (LOOP-GENERIC |FOR-LOOP-SRN$LOOP|))

:IN-THEORY
  (UNION-THEORIES '(|LOOP-INV-SRN$PROP{FOR-LOOP-SRN$STEP}|)
    (THEORY 'MINIMAL-THEORY))
  :EXPAND ((|FOR-LOOP-SRN$LOOP| N IN)))

:RULE-CLASSES NIL)
Inv Holds on Input, with type hyps

(DEFTHML ACL2-LOOP-INV$INV{INIT}
  (IMPLIES (ACL2-LOOP-INV$INV{PRE} IN)
   (|LOOP-INV-SRN$PROP| (INPUT1<_T_0> IN)
    (|LOOP-INV-SRN$PROP$INIT| IN)))
  :RULE-CLASSES NIL)
(DEFTML ACL2-LOOP-INV$INV
  (IMPLIES (ZERO-ARRAY$INPUT-HYPS IN)
    (ACL2-LOOP-INV$INV+ IN))
:HINTS
  ("Goal"
    :IN-THEORY
    (UNION-THEORIES '(ACL2-LOOP-INV$INV{PRE})
      (THEORY 'MINIMAL-THEORY))
    :USE (ACL2-LOOP-INV$INV$CONDITIONAL
      ACL2-LOOP-INV$INV{PRE}{HOLDS})))
:RULE-CLASSES NIL)
Loop counter = Loop bound

(DEFTHML LC$FOR-LOOP-SRN
  (IMPLIES (AND (NATP N)
                (NATP (G :LC IN))
                (<= (G :LC IN) N))
     (EQUAL (G :LC (|FOR-LOOP-SRN$LOOP| N IN)) N))
:HINTS ("Goal" :BY (:FUNCTIONAL-INSTANCE
  LOOP-GENERIC-LC
  (STEP-GENERIC |FOR-LOOP-SRN$STEP|)
  (PROP-GENERIC |LOOP-INV-SRN$PROP|)
  (LOOP-GENERIC |FOR-LOOP-SRN$LOOP|))
:IN-THEORY (THEORY 'MINIMAL-THEORY)
:EXPAND ((|FOR-LOOP-SRN$LOOP| N IN))))
Top Inv. Holds

(DEFTHM ACL2-TOP-INV$INV
  (IMPLIES (GAUSS$INPUT-HYPS IN)
    (G :ASN (ACL2-TOP-INV IN))))

:HINTS ("Goal" :IN-THEORY (DISABLE |FOR-LOOP-SRN$LOOP|)
  :USE (ACL2-LOOP-INV$INV
    LEMMA-2-ACL2-LOOP)))

• Uses several (simple) lemmas not shown here
Lemma Library

- Lemmas about LabVIEW primitives essential to automatic proofs
- Primitive definitions are disabled by default to (weakly) remove dependence upon definitions
- Currently ~80 theorems
Future Work

- Compositional Verification
  - Initial Approach done by hand
  - Use encapsulate to export diagram properties
  - Use bounded arithmetic
  - Use encapsulate for primitive definitions
  - Diagrams containing state
Conclusion

• Prototype system for verifying LabVIEW diagrams

• About a dozen (fully automatic) examples completed

• Feasibility of approach has been proven (for state-free diagrams)