Verifying LabVIEW Graphical Programs with ACL2

Matt Kaufmann
University of Texas at Austin
kaufmann@cs.utexas.edu

National Instruments (NI) consulting work, in collaboration with:

Jeff Kodosky, NI
Jacob Kornerup, NI
Grant Passmore, Univ. of Edinburgh (formerly UT Austin & NI intern)
Mark Reitblatt, UT Austin & NI intern

OVERVIEW

Talk objective: Give a sense of how we are tackling program correctness for a widely-used graphical language

- **LabVIEW**: Graphical programming environment from National Instruments (NI); $\sim$ 150,000 users
- **ACL2**: General-purpose theorem prover ("**A** **C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp")
  - Can be used to run and to verify (applicative) Lisp programs
- **Goal**: Use ACL2 to verify LabVIEW programs
  - Translate LabVIEW programs to ACL2
  - Assertion-based approach
  - Main focus to date: proving loops correct

Note: Analogous ongoing effort at AMD (see DCC'02 talk)

OVERVIEW

Talk objective: Give a sense of how we are tackling program correctness for a widely-used graphical language

- **LabVIEW**: Graphical programming environment from National Instruments (NI); $\sim$ 150,000 users
- **ACL2**: General-purpose theorem prover ("**A** **C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp")
  - Can be used to run and to verify (applicative) Lisp programs
- **Goal**: Use ACL2 to verify LabVIEW programs
  - Translate LabVIEW programs to ACL2
  - Assertion-based approach
  - Main focus to date: proving loops correct

Note: Analogous ongoing effort at AMD (see DCC'02 talk)

OVERVIEW

Talk objective: Give a sense of how we are tackling program correctness for a widely-used graphical language

- **LabVIEW**: Graphical programming environment from National Instruments (NI); $\sim$ 150,000 users
- **ACL2**: General-purpose theorem prover ("**A** **C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp")
  - Can be used to run and to verify (applicative) Lisp programs
- **Goal**: Use ACL2 to verify LabVIEW programs
  - Translate LabVIEW programs to ACL2
  - Assertion-based approach
  - Main focus to date: proving loops correct

Note: Analogous ongoing effort at AMD (see DCC'02 talk)

OVERVIEW

Talk objective: Give a sense of how we are tackling program correctness for a widely-used graphical language

- **LabVIEW**: Graphical programming environment from National Instruments (NI); $\sim$ 150,000 users
- **ACL2**: General-purpose theorem prover ("**A C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp")
  - Can be used to run and to verify (applicative) Lisp programs
- **Goal**: Use ACL2 to verify LabVIEW programs
  - Translate LabVIEW programs to ACL2
  - Assertion-based approach
  - Main focus to date: proving loops correct

Note: Analogous ongoing effort at AMD (see DCC'02 talk)

OVERVIEW

Talk objective: Give a sense of how we are tackling program correctness for a widely-used graphical language

- **LabVIEW**: Graphical programming environment from National Instruments (NI); $\sim$ 150,000 users
- **ACL2**: General-purpose theorem prover ("**A C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp")
    - Can be used to run and to verify (applicative) Lisp programs
- **Goal**: Use ACL2 to verify LabVIEW programs
    - Translate LabVIEW programs to ACL2
    - Assertion-based approach
    - Main focus to date: proving loops correct

Note: Analogous ongoing effort at AMD (see DCC'02 talk)

OVERVIEW

Talk objective: Give a sense of how we are tackling program correctness for a widely-used graphical language

- **LabVIEW**: Graphical programming environment from National Instruments (NI); $\sim$ 150,000 users
- **ACL2**: General-purpose theorem prover ("**A** **C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp")
  - Can be used to run and to verify (applicative) Lisp programs
- **Goal**: Use ACL2 to verify LabVIEW programs
  - Translate LabVIEW programs to ACL2
  - Assertion-based approach
  - Main focus to date: proving loops correct

Note: Analogous ongoing effort at AMD (see DCC'02 talk)

TALK OUTLINE

- Overview
- A program (our running example)
- Verification flow
- Example theorem
- Highlights of approach
- Example illustrating library development
- Summary
- Future work

A PROGRAM (OUR RUNNING EXAMPLE)

We consider a LabVIEW program, `gauss`:

- Input: k, a natural number
- Output: sum of the integers from k down to 1

***** DEMO *****

VERIFICATION FLOW

1. Run some tests.
2. Run the graph parser to produce a textual graph representation
3. Run the translator on that textual graph to produce ACL2 code from a LabVIEW program:
   - `gauss-fns.lisp` — function definitions
   - `gauss-work.lisp` — proof file, user-editable
   - `gauss.lisp` — top-level theorem
4. *Certify* these files (*“books”*) with ACL2, automatically if possible

***** DEMO *****

VERIFICATION FLOW

1. Run some tests.
2. Run the graph parser to produce a textual graph representation
3. Run the translator on that textual graph to produce ACL2 code from a LabVIEW program:
   - `gauss-fns.lisp` — function definitions
   - `gauss-work.lisp` — proof file, user-editable
   - `gauss.lisp` — top-level theorem
4. *Certify* these files (*“books”*) with ACL2, automatically if possible

***** DEMO *****

VERIFICATION FLOW

1. Run some tests.
2. Run the graph parser to produce a textual graph representation
3. Run the translator on that textual graph to produce ACL2 code from a LabVIEW program:
    - `gauss-fns.lisp` — function definitions
    - `gauss-work.lisp` — proof file, user-editable
    - `gauss.lisp` — top-level theorem
4. *Certify* these files (“*books*”) with ACL2, automatically if possible

***** DEMO *****

VERIFICATION FLOW

1. Run some tests.
2. Run the graph parser to produce a textual graph representation
3. Run the translator on that textual graph to produce ACL2 code from a LabVIEW program:
   - `gauss-fns.lisp` — function definitions
   - `gauss-work.lisp` — proof file, user-editable
   - `gauss.lisp` — top-level theorem
4. *Certify* these files (*“books”*) with ACL2, automatically if possible

***** DEMO *****

VERIFICATION FLOW

1. Run some tests.
2. Run the graph parser to produce a textual graph representation
3. Run the translator on that textual graph to produce ACL2 code from a LabVIEW program:
   - `gauss-fns.lisp` — function definitions
   - `gauss-work.lisp` — proof file, user-editable
   - `gauss.lisp` — top-level theorem
4. *Certify* these files (*“books”*) with ACL2, automatically if possible

***** DEMO *****

HEY, WAIT A MINUTE!

VERIFICATION COMPLETED!

(Maybe we’ll take a quick peek at gauss-work.lisp.)

EXAMPLE THEOREM

Top-level file generated by our verification process:

```
(IN-PACKAGE "ACL2")

; Translation of program to ACL2 functions:
(INCLUDE-BOOK "gauss-fns")

; Include proof file (user-editable); ignore when
; reading this final result for logical content.
(LOCAL (INCLUDE-BOOK "gauss-work"))

(SET-ENFORCE-REDUNDANCY T)

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

HIGHLIGHTS OF APPROACH

I'll go through the generated code and illustrate some key points:

- Modeling dataflow programs with ACL2 functions
- Modeling loops with recursion
- Proving correctness of loops: a generic VCG-like approach

Modeling dataflow programs with ACL2 functions

Note that the translation to functions is mechanical and (at least at a high level) straightforward.
Here is a snippet from file `gauss-fns.lisp` – just a quick look here:

```
(DEFUN-N |_N_10| (IN)
  (S* :ASN (EQUAL?-0<_T_2> IN)))

(DEFUN-ASN ACL2-TOP-INV (IN)
  (|_N_10| (S* :|_T_1| (INPUT1<_T_0> IN)
               :|_T_2| (ACL2-LOOP<_T_6> IN))))
```

Note that our translation supports evaluation in ACL2.

Modeling dataflow programs with ACL2 functions

Note that the translation to functions is mechanical and (at least at a high level) straightforward.
Here is a snippet from file `gauss-fns.lisp` – just a quick look here:

```
(DEFUN-N |_N_10| (IN)
  (S* :ASN (EQUAL?-0<_T_2> IN)))

(DEFUN-ASN ACL2-TOP-INV (IN)
  (|_N_10| (S* :|_T_1| (INPUT1<_T_0> IN)
               :|_T_2| (ACL2-LOOP<_T_6> IN))))
```

Note that our translation supports evaluation in ACL2.

Modeling loops with recursion

I'll talk through the following from file `gauss-fns.lisp`:

```
(DEFUN |_N_15$LOOP| (N IN)
  (DECLARE (XARGS :MEASURE (NFIX (- N (G :LC IN)))))
  (COND ((OR (>= (G :LC IN) N)
             (NOT (NATP N))
             (NOT (NATP (G :LC IN))))
         IN)
        (T (|_N_15$LOOP| N
              (S :LC (1+ (G :LC IN))
                 (|_N_15$STEP| IN))))))
```

Proving correctness of loops

- Once and for all: introduce a *generic* loop function and prove its correctness.
- Prove:
  - The actual loop invariant is true initially; and
  - The actual step function preserves the actual loop invariant.
- Conclude using ACL2's *functional instantiation* technique that the actual loop invariant holds.

EXAMPLE ILLUSTRATING LIBRARY DEVELOPMENT

***** DEMO (zeroing out an array) ***** — if time

SUMMARY

We have a mechanical approach to:

- translating LabVIEW diagrams into ACL2; and
- verifying loops with automated support.

Just an aside: The translator is written in ACL2. “Guard checking” helped catch bugs!

FUTURE WORK

Our approach works on small examples, but there's lots more to do.

- Move away from semantics of unbounded integers, and in general support more data types.
- Handle state: limited I/O and global variables.
- Develop graphical interface: e.g., remove proved assertion wires.
- Improve support for modularity, building on a nested loop example already worked.
- Complete handling of unbounded while-loops.
- Support verification of timing properties for LabVIEW on FPGAs.
- More examples may lead us to use ACL2's hook for connecting other proof tools.
- Goal: NI Labs (`http://www.ni.com/labs/`)