Formal Verification of LabVIEW Programs Using the ACL2 Theorem Prover

Matt Kaufmann
Dept. of Computer Sciences,
University of Texas at Austin
kaufmann@cs.utexas.edu

Jacob Kornerup
National Instruments, Inc.,
jacob.kornerup@ni.com

Mark Reitblatt
Dept. of Computer Sciences,
University of Texas at Austin
National Instruments, Inc.
mark@reitblatt.com

ABSTRACT

The LabVIEW system is based on a graphical dataflow language, and is widely used for data acquisition, instrument control and industrial automation. This paper presents a methodology for annotating LabVIEW programs with their specifications, translating those annotated programs into ACL2, and proving the translated specifications with ACL2. Our system supports verification of inductive invariants of bounded loops as well as assertions about straight-line code. Our verification methodology supports the user by generating a highly structured set of proof obligations, many or all of which are discharged automatically. This methodology makes extensive use of hints to support scalability, including careful theory control as well as functional instantiation that avoids explicit use of induction. We describe the design, applicability and limitations of the framework. We also present several examples demonstrating our approach.

1. INTRODUCTION

LabVIEW is a widely used product of National Instruments, Inc. that supports data collection, instrument control and industrial automation. Its dataflow language is similar in semantics to a functional language, with write-once dataflow *wires* taking the place of local variables and a very simple control flow. The full LabVIEW language contains several decidedly non-functional features such as synchronization primitives and global variables. We have chosen to work with a restricted yet well featured subset of LabVIEW's language that contains only purely functional elements.

LabVIEW programs are graphical diagrams called VIs (Virtual Instruments). Wires carry data from left to right between *nodes*, which are inputs, outputs, or instances of functional units. When diagram input nodes (actual parameters) obtain their values, their outputs are carried down wires to other nodes, which in turn fire once all of their inputs have been set. Figure 1 illustrates how this works. Since there are no inputs to the diagram and the constant nodes have no inputs, they are free to fire by pushing their value, 1, to their output wires. The '+' node then obtains its two inputs, 1 and 1, and adds them to produce the value 2 on its output wire, which in turn "feeds" the output of the diagram, labeled "`Output`".

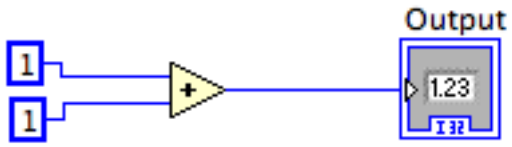


Figure 1: Simple LabVIEW diagram

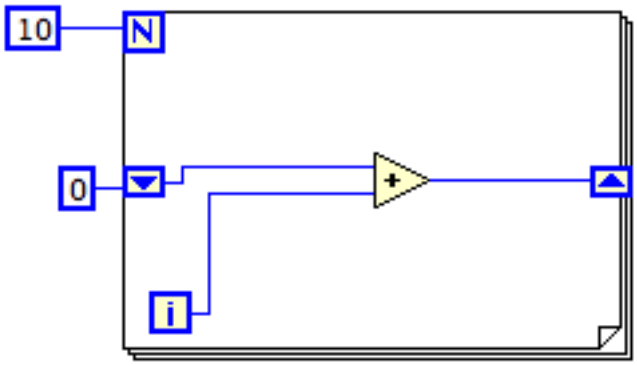
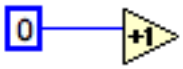


Figure 2: LabVIEW diagram with a for-loop

Functional units can be one of LabVIEW's built-in primitives or other LabVIEW diagrams called *sub-VIs*, instances of which are embedded in the parent diagram. Control structures such as for-loops, as shown in Figure 2, are also functional units,. All functional units, including control flow structures, operate by waiting for all of their inputs to be set before they begin executing. Local state between iterations is stored in *shift registers*, the boxes with arrows in

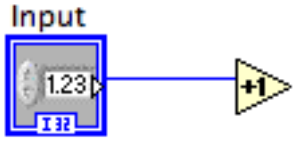



```
(defun-n constant[0]-0 (in)
  (S* :_T_0 0))

(defun-w constant[0]-0<_T_0> (in)
  (G :_T_0 (constant[0]-0 in)))

(defun-n increment-0 (in)
  (S* :_T_1 (1+ (constant[0]-0<_T_0> in))))
```

Figure 3: Simple LabVIEW diagram translation




```
(defun-w input<_T_0> (in)
  (G :input in))

(defun-n increment-0 (in)
  (S* :_T_1 (1+ (input<_T_0> in))))
```

Figure 4: LabVIEW diagram with input

Figure 2. In our particular example, the *loop bound* is 10, so there are 10 iterations of the loop, each adding the current value of the *loop counter* `i` to the accumulated sum that is stored in the shift register (initially, 0). The output is thus the sum $0 + 1 + \ldots + 9$.

The rest of this paper begins with a discussion of our translation methodology for programs and specifications. We then explain our verification infrastructure, including automatic generation of lemmas, by way of a running example. Finally, we discuss future work and conclude.

2. TRANSLATION

We begin by describing our translation of LabVIEW programs to ACL2 functions. Then we explain our methodology for extending programs with specifications.

2.1 LabVIEW Model

The translation is kept as simple as possible by closely mimicking the original data-flow structure. The input LabVIEW diagram's nodes and wires are represented by ACL2 functions. A node takes a record structure [4] representing the input to the whole diagram and return a record, with keys representing its output terminals (names) bound to the output values. Wires take a diagram input record and return the value extracted from their driving node.

In Figure 3 the second ACL2 function represents the wire holding the value of the `_T_0` output terminal of the first constant node representing the constant 0. Since there are no inputs to this diagram, the `in` record is actually ignored. Contrast that with Figure 4, which contains one diagram input. Note that `(G` k r`)` gets the value of key k in record r, while `(S*` k_0 v_0 ... k_n v_n`)` creates the record that sets the value of key k_i to value v_i for i from 0 to n.

For-loops and other iterative structures present an interesting challenge. In LabVIEW each control structure is split into two nodes, an "inner" Self Reference Node (SRN) and an outer node, roughly corresponding to a function definition and a corresponding function call, respectively. The outer node handles the initialization of structure inputs and the binding of structure outputs. In Figure 2 the outer node reads the initial values for the shift register and the loop bound, `N`, and initializes the loop counter, `i`. It then checks to see if the loop counter is less than the loop bound and, if so, runs the SRN with the single wire bound to the value in the shift register. At the next iteration it increments the loop counter by 1 and checks the bound, and, if the bound check passes, it reads the last iteration's value out of the shift register and runs the SRN once again, but with the wire holding the new shift register value. Once the bound check fails the outer node binds the output wire to the latest shift register value and stops running.

We have chosen to model these structures using four separate ACL2 functions. In Figure 5 we show a translation of the diagram in Figure 2; `for-loop-srn` is the SRN node of `for-loop`. `for-loop` calls `for-loop-srn$loop` with arguments with the loop bound and an initial environment (set up by `for-loop-srn$loop$init`). `for-loop-srn$loop` is a tail recursive function that models the actual iteration structure by checking the loop bound at every iteration. `for-loop-srn$step` grabs the outputs of the loop's iteration and binds the next iteration's inputs.

One interesting requirement for a translation system like this one is a human invertible mapping between the original (LabVIEW) program and the generated ACL2 functions. When working with proofs of non-trivial diagrams it is essential to be able to relate the verification items to the actual diagram in order to gain insight into the process. Unfortunately LabVIEW doesn't provide a mechanism for naming wires and some nodes, so the translator must choose an insightful yet unique name for each node. In our current (and most successful) naming scheme we name each node according to its type (such as increment or multiply) and the number of such nodes previously seen. Unfortunately it's impossible to know what order the nodes will have when they come out of the first compiler, so it still requires a little work to exactly pin the ACL2 functions to the LabVIEW diagram. Wires are named according to the (unique) node and terminal which provides its value; since multiple nodes may read from the same wire, it's impossible to name wires according to target as well as source.

It is also worth noting that LabVIEW uses standard fixed-size data types such as 16 or 32 bit integers, but we implement arithmetic in our model using ACL2's arithmetic functions with idealized integers. This allows us to use ACL2's

```
(DEFUN FOR-LOOP-SRN$STEP (IN)
  (S :|_T_4| (G :|_T_1| (|_N_5| IN)) IN))

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

(DEFUN FOR-LOOP-SRN$LOOP$INIT (IN)
  (S* :LC 0
      :|_T_2| (CONSTANT[10]-1<_T_0> IN)
      :|_T_4| (CONSTANT[0]-0<_T_0> IN)))

(DEFUN-N FOR-LOOP (IN)
  (FOR-LOOP-SRN$LOOP (CONSTANT[10]-1<_T_0> IN)
     (FOR-LOOP-SRN$LOOP$INIT IN)))
```

Figure 5: ACL2 translation of figure 2

powerful arithmetic libraries, but puts a limitation on our claim to be verifying actual LabVIEW semantics. That is, our diagrams are verified against an idealized semantics, and one needs an implicit assumption about the absence of overflow in order for the idealized semantics to agree with the actual semantics.

2.2 Specifications

It is important to give the programmer an easy mechanism for specifying functional properties that is not too difficult to translate into the underlying formalism. Because we already translate LabVIEW directly into ACL2, we chose to use designated boolean valued blocks of LabVIEW code as specifications. This choice presents several benefits:

- Specifications are straightforward for LabVIEW programmers to write
- Specifications are given the same semantics as the LabVIEW program itself
- Specifications can be checked dynamically during LabVIEW runs

However, sometimes it is not straightforward to write the intended specification in LabVIEW, for example when LabVIEW's lack of recursion would require one to use a loop in the specification. For these instances we have implemented a "stub node" that allows programmers to refer to an arbitrary ACL2 function in their specifications.

Assertions for simple straight-line diagrams can be specified by a single assertion box placed on a diagram. Loop specifications take the form of two assertion blocks; one is a loop invariant and the other is the actual desired loop specification. We defer further discussion of the graphical representation of specifications to the following section, where we show how this works for a loop.

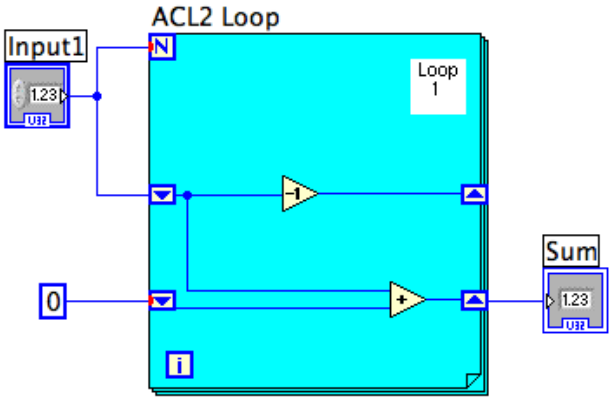


Figure 6: LabVIEW diagram "gauss"

3. VERIFICATION PROCEDURE

We explain our verification procedure by way of an example. First, we explain the example. Then we describe the files generated by the translator. Next we outline the lemmas we generate and how they fit together. We conclude with a brief discussion of library development.

3.1 A Running Example

Our example diagram is named "gauss". This diagram takes a natural number input, N, and returns the result of adding all the natural numbers from N down to 1. See Figure 6. While this program is similar to the one in Figure 2, it counts down instead of up because that presents more of a challenge: unlike that previous diagram, here the loop invariant will not simply be the result of instantiating the top-level property, with the loop bound replaced by the loop counter.

Our top-level specification is that this sum equals $N(N+1)/2$. The diagram can be extended for this specification as shown in Figure 7, to contain two top-level blocks: the for-loop block and a block for the top-level specification. The output of the for-loop block (representing the above sum) is an input to the invariant block, whose output is the Boolean generated by an equality node with two inputs, essentially: that sum output, and a node whose output equals $N(N+1)/2$.

As with traditional program verification, the user supplies a loop invariant, i.e., a property that is true at loop entry, is preserved by each iteration through the loop, and is sufficient to prove the top-level specification. Figure 7 shows a block for the loop invariant, inside the for-loop block.

Here is a textual representation of that inner block, where: c denotes the value being counted down, which is initially N and is decremented by 1 at each iteration, visually represented as the upper shift register in the diagram; and i denotes the loop counter, which (as for every for-loop) is initially 0 and is incremented by 1 at each iteration.

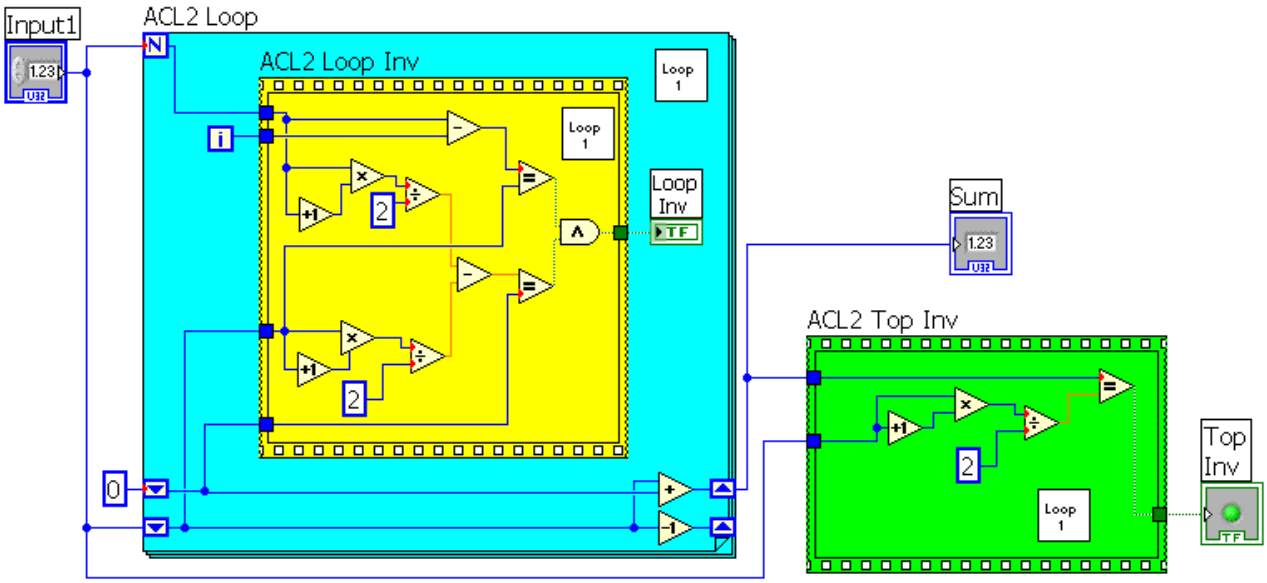


Figure 7: LabVIEW diagram "gauss", annotated

$$\begin{aligned} sum &= N(N+1)/2 - c(c+1)/2 \\ &\& \\ c &= N - i \end{aligned}$$

Before leaving this introduction to our example, we say a word about why this loop invariant implies the top-level assertion that $sum = N(N+1)/2$. A critical observation is that as the loop exits, i has reached N. Thus $c = 0$ at exit, and the equation for sum above reduces to the desired equation. We expect this observation, namely that the final value of the loop counter is equal to the loop bound, to be useful quite generally. We return to this point at the end of Section 3.3.

3.2 Generated Book Structure

We generate three files for an annotated diagram that are intended to become certified books. For our example, these files are included in the supporting materials for this paper.[1].

The *functions book* (`gauss-fns.lisp`) contains the actual translation of the LabVIEW diagram into ACL2, as described above.

The *top book* (`gauss.lisp`), shown below, includes the functions book (by `include-book`) and states the main result, that the output of the top-level specification block is `T`. Here, `:ASN` is the (unique) key of the record `(ACL2-TOP-INV IN)` returned by the top-level invariant block on an arbitrary input, `IN`, and `GAUSS$INPUT-HYPS` states input type hypotheses derived from the diagram, in this case, `(NATP (G :INPUT1 IN))`.

```
(IN-PACKAGE "ACL2")

(INCLUDE-BOOK "gauss-fns")

(LOCAL (INCLUDE-BOOK "gauss-work"))

(SET-ENFORCE-REDUNDANCY T)

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

The local `include-book` form indicates the third book, the *work book*: `gauss-work.lisp`. Because this book is included only locally in the top book, certification of the top book guarantees that the final theorem is a logical consequence of the definitions in the functions book alone [3]. Thus, the user has total freedom to edit the work book without compromising the validity of the verification effort, provided that ultimately the top book certifies. Any mistakes the user left in the work file such as the use of `skip-proofs`, `defaxiom`, or trust tags will of course prevent the work book, and hence the top book, from certifying. The generation of the `set-enforce-redundancy` event above is merely for user convenience, to highlight any editing mistake the user may have made in the final theorem of the work book.

Of course, our goal is to automate the creation of the work book so as to minimize the amount of user effort. We turn now to describing the layout of the work book, which facilitates automatic proof of the "easy" parts, while presenting appropriate problem-specific proof obligations that may require user effort to prove. In this particular example no user effort is required: `gauss-work.lisp` certifies automatically.

3.3 Generated Proof Events

In this section we describe our approach to verifying properties of for-loops. (The verification of loop-free code is consid-

[1]In the discussion below, globally substitute for "gauss" to get filenames for diagrams with other names.

erably simpler.) To help the user we generate a highly structured proof scaffolding, as an orchestrated series of lemmas leading to the proof of the user-supplied inductive invariant and then, from that, the top-level specification. In particular we avoid explicit proof by induction, by using functional instantiation [1] based on a generic theory where the induction is done using an encapsulated inductive invariant.

The proof strategy may be thought of in two parts. First, show that the loop invariant is preserved by a single step and hence by the loop, so since it is true initially, therefore the loop invariant holds. Then show that the top-level specification holds by providing that loop invariant theorem as a `:use` hint.

We now give some details on how the above strategy is realized in a work book by focusing on file `gauss-work.lisp`. Note that the structure is the same for any for-loop. Although these general sorts of methods are well-known, the trick here is to generate suitable lemmas for dealing with details such as type information and the final value of the loop counter, while providing automated proof for all but the key problem-specific pieces. Many such lemmas are omitted here, but again, they may be found in file `gauss-work.lisp` included with the supporting materials to this paper.

1. Extend the loop invariant.

Our loop invariant extends the loop invariant provided by the user, which is represented by the third conjunct below. The first conjunct represents type information derived from the LabVIEW diagram (here, the input is a natural number). The second conjunct equates variable `N` with the loop bound (input name `_T_3` in this case).

```
(DEFUN |_N_33$PROP| (N IN)
  (DECLARE (IGNORABLE N))
  (AND (|_N_33$HYPS| IN)
       (EQUAL N (G :|_T_3| IN))
       (G :ASN (ACL2-LOOP-INV IN))))
```

2. The loop invariant is preserved by taking a step.

This lemma may require user interaction (as noted in a comment) such as proving lemmas, including books, and providing hints. But for simple examples the proof can be fully automatic, as it is in the present example. Note the use of function `(S` k v r`)`, which creates the modification of record r that binds key k to value v, here reflecting the incrementing by 1 of the loop counter when taking one step of the loop.

```
; USER ASSISTANCE MAY BE REQUIRED:
(DEFTHMDL |_N_33$PROP{_N_8$STEP}|
  (IMPLIES (AND (NATP (G :LC IN))
                (< (G :LC IN) N)
                (|_N_33$PROP| N IN))
           (|_N_33$PROP| N
            (S :LC (1+ (G :LC IN))
               (|_N_8$STEP| IN)))))
```

3. The loop invariant is preserved by running the full SRN (uninstantiated) loop.

Encapsulated functions `step-generic` and `prop-generic` are introduced in book `generic-loop-inv.lisp` to satisfy the following constraint:

```
(defthm prop-generic-step
   (implies (and (natp n)
                 (natp (g :lc in))
                 (< (g :lc in) n)
                 (prop-generic n in))
            (prop-generic n (s :lc (1+ (g :lc in))
                               (step-generic in)))))
```

Also in the above book, one finds a straightforward definition of a generic loop function and a simple inductive consequence of that constraint:

```
(defun loop-generic (n in)
  (declare (xargs :measure (nfix (- n (g :lc in)))))
  (cond ((or (not (natp n))
             (not (natp (g :lc in)))
             (>= (g :lc in) n))
         in)
        (t (loop-generic n
                         (s :lc (1+ (g :lc in))
                            (step-generic in))))))

(defthm loop-generic-thm
  (implies (and (natp n)
                (natp (g :lc in))
                (prop-generic n in))
           (prop-generic n (loop-generic n in)))
  :hints (("Goal" :induct t)))
```

An important lemma for a specific diagram can then be proved automatically by functional instantiation, with an `:in-theory` hint that provides a small, controlled theory to promote scalability. In our example that lemma is as follows.

```
(DEFTHML |_N_33$PROP{_N_8}|
  (IMPLIES (AND (NATP N)
                (NATP (G :LC IN))
                (|_N_33$PROP| N IN))
           (|_N_33$PROP| N (|_N_8$LOOP| N IN)))
  :HINTS
  (("Goal" :BY (:FUNCTIONAL-INSTANCE
                LOOP-GENERIC-THM
                (STEP-GENERIC |_N_8$STEP|)
                (PROP-GENERIC |_N_33$PROP|)
                (LOOP-GENERIC |_N_8$LOOP|))
           :IN-THEORY
           (UNION-THEORIES '(|_N_33$PROP{_N_8$STEP}|)
                           (THEORY 'MINIMAL-THEORY))
           :EXPAND ((|_N_8$LOOP| N IN))))
  :RULE-CLASSES NIL)
```

4. The loop invariant holds of the outer (instantiated) loop, assuming loop input types hold.

The following theorem says the loop invariant is true when entering the outer loop, that is, is true initially as the loop sits in the environment provided by the top-level diagram (i.e., with loop inputs provided by that diagram):

```
(DEFTHML ACL2-LOOP-INV$INV{INIT}
  (IMPLIES (ACL2-LOOP-INV$INV{PRE} IN)
           (|_N_33$PROP| (INPUT1<_T_0> IN)
            (|_N_33$PROP$INIT| IN)))
  :RULE-CLASSES NIL)
```

The following predicate asserts that the loop invariant holds for the outer loop.

```
(DEFUN ACL2-LOOP-INV$INV+ (IN)
  (DECLARE (XARGS :NORMALIZE NIL))
  (|_N_33$PROP| (INPUT1<_T_0> IN)
   (|_N_8$LOOP| (INPUT1<_T_0> IN)
    (|_N_8$LOOP$INIT| IN))))
```

The lemmas `ACL2-LOOP-INV$INV{INIT}` (just above) and `|_N_33$PROP{_N_8}|` (above, SRN loop invariant preservation) are key for proving that the outer loop holds, as stated with above predicate `ACL2-LOOP-INV$INV+`, under the assumption that the loop inputs satisfy their expected type hypotheses.

```
(DEFTHML ACL2-LOOP-INV$INV$CONDITIONAL
         (IMPLIES (ACL2-LOOP-INV$INV{PRE} IN)
                  (ACL2-LOOP-INV$INV+ IN))
         :HINTS ...)
```

5. The loop invariant holds of the outer (instantiated) loop, without assumptions on types.

Here we show that the necessary type hypotheses do indeed hold. Again, user assistance may be required, though one expects that to be rare in simple cases, like this example.

```
(DEFTHML ACL2-LOOP-INV$INV{PRE}{HOLDS}
  (IMPLIES (GAUSS$INPUT-HYPS IN)
           (AND (NATP (INPUT1<_T_0> IN))))
  :RULE-CLASSES NIL)
```

And now we can prove that the loop invariant holds assuming only the input type hypotheses for the top-level diagram.

```
(DEFTHML ACL2-LOOP-INV$INV
  (IMPLIES (GAUSS$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)
```

6. The top-level invariant holds.

It remains only to prove the top-level assertion. A couple of automatically generated lemmas are proved automatically (in a scalable way, with small theories), and the top-level assertion then follows, though user assistance may be required.

```
(DEFTHM ACL2-TOP-INV$INV
  (IMPLIES (GAUSS$INPUT-HYPS IN)
           (G :ASN (ACL2-TOP-INV IN)))
  :HINTS (("Goal" :IN-THEORY (DISABLE |_N_8$LOOP|)
           :USE (ACL2-LOOP-INV$INV
                 LEMMA-2-ACL2-LOOP))))
```

The proof goes through automatically in the `gauss` example. But the proof requires the following lemma in the work book that is both generated and proved automatically. Recall from the end of Section 3.1 that for this example, it is critical to know that the final value of the loop counter is equal to the loop bound. That theorem is included in our generic theory, so we get a fast, reliable proof for our example using functional instantiation.

```
(DEFTHML LC$_N_8
  (IMPLIES (AND (NATP N)
                (NATP (G :LC IN))
                (<= (G :LC IN) N))
           (EQUAL (G :LC (|_N_8$LOOP| N IN)) N))
  :HINTS (("Goal" :BY (:FUNCTIONAL-INSTANCE
                       LOOP-GENERIC-LC
                       (STEP-GENERIC |_N_8$STEP|)
                       (PROP-GENERIC |_N_33$PROP|)
                       (LOOP-GENERIC |_N_8$LOOP|))
           :IN-THEORY (THEORY 'MINIMAL-THEORY)
           :EXPAND ((|_N_8$LOOP| N IN)))))
```

3.4 Library

An important part of a verification framework for a language is an expansive set of theorems about that language's primitives. Since for the most part we translate LabVIEW primitives into new functions that we have defined, it is necessary to prove many basic theorems before any progress can be made. For example, we have to prove that LabVIEW's `array-reverse` is its own inverse.

It is possible that in the future our definitions of primitives may change, due either to a discovered disagreement with LabVIEW's actual semantics or perhaps the desire to increase simulation performance. To this end we keep the LabVIEW primitives' definitions disabled when proving theorems in the work file, and instead rely solely upon higher level properties proven in a library of theorems about LabVIEW primitives. The idea here is that when we are verifying diagrams, we have to lean on these properties which

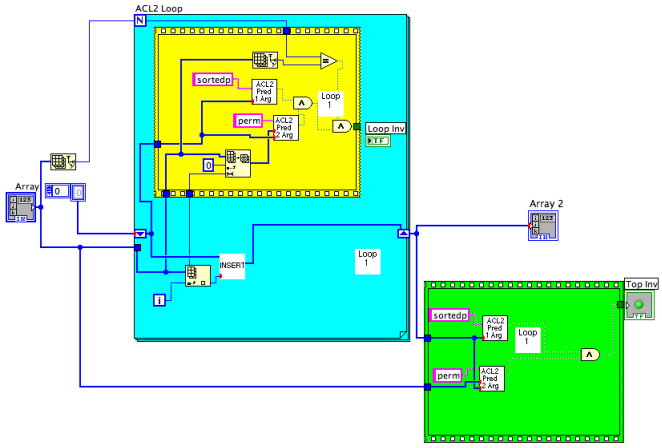


Figure 8: Insertion sort VI using insert sub-VI and ACL2 predicates

are true of the actual LabVIEW primitives instead of the specific ACL2 implementation of the primitives which may have additional properties. Of course, many of our library theorems have been discovered as we worked examples and examined failed proof output.

4. FUTURE WORK

4.1 Compositional Verification

It is generally agreed that for a verification methodology to be practical it must be able to employ some form of compositional reasoning. LabVIEW's granularity of abstraction is a sub-VI, so it is only natural to base a compositional approach around sub-VIs. Towards this end we have a preliminary methodology completed by hand. The basic idea is for a sub-VI to "export" a constrained function that corresponds to the sub-VI, along with an optional library file containing a theory built around that constrained function. When another VI in turn uses an instance of that sub-VI, we only use the exported properties of the sub-VI. By separating the desired properties of the sub-VI from the actual implementation we allow component-wise verification for even partially completed systems.

For an example of this approach, consider Figure 8. Here we have an implementation of a simple insertion sort. The real work happens in the insert sub-VI, but the high level verification of insertion sort can happen even before writing the insert sub-VI. This diagram is verified automatically under the constraints that "insert" returns a sorted list if its first argument is a sorted list and that it returns a permutation of the extension of its input list by consing, to the front, the item to be inserted. Separately we have an actual "insert" sub-VI that has these properties.

4.2 Other Additional Capabilities

We have accumulated a list of improvements that we would like to make to our methodology and tools.

As we discussed in Section 2.1, we currently support a semantics of unbounded integers for LabVIEW's bounded data types. We would like to rectify that situation, both by suitable translator modification, and with incorporation and/or development of suitable libraries.

Although there are many interesting LabVIEW programs that are purely functional, we would like to handle state, including limited I/O and global variables. We have done some very simple proofs but haven't yet automated the translation or done anything of size. Not surprisingly, handling state seems to pose quite a challenge!

There is more we can do to support the work style of LabVIEW users, who tend to work in its graphical interface. For example, proved assertions might be removed or recolored.

We handle for-loops with a fixed iteration bound. We intend to support LabVIEW *while-loops* as well, perhaps using `defpun` [5] to model unbounded recursion.

Since LabVIEW runs on FPGAs, which suggests the potential for formal verification of timing properties for the generated hardware designs.

As we work more complex examples we expect to find additional ways to improve proof support. For example, we may employ ACL2's clause-processor hook [2] to connect proof tools based on efficient decision procedures.

5. CONCLUSION

We have described a system for translating annotated LabVIEW diagrams into files containing ACL2 functions and theorems. We have employed this system to work about a dozen examples. We have extended our library of ACL2 functions for LabVIEW primitives as needed by the proofs, and proved useful facts about these LabVIEW programs. All their proofs are now done automatically by ACL2.

The general methods used in this paper are well-known. Our technical contribution is to generate appropriate lemmas automatically that deal with details such as type information and the final value of the loop counter, and to generate suitable hints to provide automation that scales to large problems, while providing fast automatic proof for all but the key problem-specific pieces. Our contribution is also to connect formal verification to a widely-used programming language with some 150,000 users world-wide.

We continue to extend the methodology as we gain more experience. For example, in earlier versions of the framework, loop invariants were placed outside the loop. This presented many complications involving the appropriate relation of the inputs to the invariant block to the actual loop inputs. Moving the invariant inside the loop reduced the complexity of the translation and simplified the task of writing invariants in LabVIEW.

This project is in a relatively early stage. We have outlined some of the many areas in which to extend our efforts. In particular, our investigation of hierarchy is important for supporting scalability to large collections of components. This includes the study of proof reuse and linking together proofs of subcomponents in the proof of properties of a component.

Acknowledgements

We thank Jeff Kodosky, the originator of LabVIEW, for his guidance and inspiration, and also Warren Hunt and J Moore for their efforts in getting this project going as well as J Moore's guidance in advising Mark. We are especially grateful to Grant Passmore for his significant contributions in the early phases, while an intern at National Instruments, where he worked out a first approach to using ACL2 to verify LabVIEW programs. Preparation of this paper was supported in part by DARPA and the National Science Foundation under Grant No. CNS-0429591.

6. REFERENCES

[1] R. Boyer, D. Goldschlag, M. Kaufmann, and J. S. Moore. Functional instantiation in first-order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[2] M. Kaufmann, J. Moore, S. Ray, and E. Reeber. Integrating external deduction tools with ACL2. In C. Benzmueller, B. Fischer, and G. Sutcliffe, editors, *Proceedings of the 6th International Workshop on Implementation of Logics (IWIL 2006)*, volume 212 of *CEUR Workshop Proceedings*, pages 7–26, 2006. to appear in the Journal of Applied Logic.

[3] M. Kaufmann and J. S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[4] M. Kaufmann and R. Sumners. Efficient rewriting of data structures in acl2. In *Proceedings of the Third International Workshop on the ACL2 Theorem Prover and its applications*, Grenoble, France, April 2002.

[5] P. Manolios and J. S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.