Thesis Draft

Mark Reitblatt

April 30, 2009

Abstract

The purpose of this thesis is to present a prototype system for the verification of LabVIEW programs using ACL2. LabVIEW is a graphical data-flow programming language commonly used in data acquisition and control application. ACL2 is a programming language, formal logic and theorem prover that has seen broad use in the verification of industrial hardware and software systems.

I will present an ACL2 model of LabVIEW programs, a system for translating LabVIEW programs into their ACL2 models, a method for annotating LabVIEW programs with specifications, a methodology for the verification of the ACL2 model of a LabVIEW program with respect to its specification and a library of theorems about our ACL2 models of LabVIEW primitives designed to increase the ease and accessibility of verification. I will also present several small examples of verified LabVIEW programs to demonstrate the application of the system.

1 Introduction

1.1 Background

The construction of correct, reliable artifacts is the primary focus of any engineering discipline. In most mature engineering disciplines, design simulation and verification are routine steps in the production of any artifact, serving to guarantee basic quality and reliability. Programming, however, has not yet reached a level of maturity expected of most engineering disciplines, with design verification being far from routine. Current industrial best practices consist primarily of targeted unit testing, and even this is often ignored in practice.

There is a myriad of techniques that can be used to improve the reliability of software products ranging from runtime error checking to type systems to full verification and a wide spectrum in-between. The specific goals and practical difficulties vary in each approach, but the overall intention is the same: to help the programmer to construct error-free programs.

Formal verification is one approach to program verification that has seen little uptake in the software world outside of safety critical applications (avionics, life support systems etc.). Formal verification promises, in theory, the possibility of full functional verification; that is, the demonstration that a program meets all of its requirements. In practice, formal verification ranges from difficult to impossible. Often times formal tools are not even available for the project at hand and, when they are, they require extensive and specialized expertise rarely found on general software development teams. Furthermore, specifications are overwhelmingly provided as informal instructions. Indeed, the practice of procuring and writing unambiguous specifications is a major problem in and of itself!

It is rarely cost effective or even particularly desirable to formally verify the whole of a software product, especially when verification would cost a company a competitive edge by delaying the shipping date. Having a software product that continues to perform correctly four years down the road does the producer little good if they went out of business in two years because a competitor beat them to the market. In order to increase the desirability and practicality of formal verification it is necessary to integrate verification into existing development tools as well as to reduce the learning curve required for interacting with and understanding the verification tools themselves.

Figure 1: Parallel execution paths

1.2 Project History

Jeff Kodosky, inventor of LabVIEW, has had a longstanding interest in integrating formal verification into the LabVIEW development environment. Towards this end, he consulted with several researchers from the department of Computer Sciences at the University of Texas at Austin (UT) over the course of several years. As a result, Grant Passmore, then an undergraduate in the department, was hired by National Instruments as an intern in the summer of 2005 in an initial attempt to develop a system to verify properties of LabVIEW diagrams. His first project used a theorem prover custom written for LabVIEW. This first attempt showed promise and led to the verification of several diagrams, but ran into difficulties as time went on. Continuing with the momentum of this progress, Matt Kaufmann, a Senior Research Scientist at UT and co-author of ACL2, was hired in 2007 as a contractor to develop an approach to verifying LabVIEW using ACL2. By the end of the summer, Grant had developed an initial system to translate LabVIEW into ACL2 and Matt had hand-verified 2 different versions of a LabVIEW diagram. Since Grant was admitted to the PhD program at the University of Edinburgh for Fall 2007, the author was hired to continue the project with Matt Kaufmann continuing as a contractor.

1.3 LabVIEW

LabVIEW is a graphical data-flow language commonly used for data acquisition, instrument control and industrial automation. A LabVIEW program (called a VI) is expressed as a diagram composed of nodes representing actions and data-flow wires that funnel inputs and outputs to and from those nodes. Data flows along the wires from left to right. Action nodes have a number of input terminals on the left and output terminals on the right. In general there is no restriction that all inputs to a node be represented through wires (e.g. non-deterministic or stateful nodes), but the subset of LabVIEW that we are concerned with here contains only pure function nodes so all inputs are present. Nodes fire in arbitrary order as soon as all of their inputs are available; in particular, constant nodes have no inputs and can fire at any time. Once the nodes compute their output they in turn pass it along output wires to waiting nodes. Note that since all nodes are functional, the execution order doesn't matter.

LabVIEW includes a large number of built-in primitive nodes including arithmetic and array operations. There is also the capability to use other diagrams as subroutines by embedding them as subVI nodes. These nodes function exactly like primitive nodes, but are treated differently in our translation, which will be explained in Section 2.5.

1.4 ACL2

Roughly speaking, ACL2 is an applicative (purely functional) subset of Common Lisp which has been formalized as a mathematical logic and coupled with a full strength theorem prover. It is far beyond the scope of this paper to give a proper introduction to ACL2, but we cover the parts essential to understanding the work done here, with the intention that readers unfamiliar with ACL2 will be able to follow the discourse with reference back to this section.

The first, and most noticeable, aspect of ACL2 is that it uses Lisp s-expression syntax with prefix function application. That is, the essential syntactic element of ACL2 is the parenthesis, '(' and ')'. In mathematical notation, one would write `f(x,y)` to denote the application of function `f` to the arguments `x,y`. In ACL2 we write `(f x y)` to denote the same application. This gets a little confusing as we write traditionally infix operators such as `+` as prefix. To wit, we would write `x+y` as `(+ x y)`.

The next important ACL2 elements are lists. In ACL2, as in all Lisps, the list is the primary data structure. Ordered pairs are constructed with the `cons` operator. `(cons a b)` forms the tuple `(a . b)`. We recursively define lists as either `nil` (which denotes both the empty list and false), or `(cons a lst)` where `lst` is a list. So, we define the list `(0 1 2)` by `(cons 0 (cons 1 (cons 2 nil)))`. For notation, if the

second element of a `cons` is a list then we drop the dot. So, `(cons 1 2)` = `(1 . 2)` and `(cons 1 nil)` = `(1)`. Note that since ACL2 is functional and lacks pointers, all lists are finite and acyclic.

Lists are deconstructed using one of the two list operators, `car` and `cdr`. These operators are defined as `(car (cons a b))` = `a` and `(cdr (cons a b))` = `b`. Another way of thinking of `car` and `cdr` is that `car` returns the first item of the list (`head` in some languages) and `cdr` returns the list minus the first element (`tail`).

1.5 Encapsulate

As a first order logic, ACL2 doesn't allow higher order functions or theorems. However, ACL2 has weak "second order" functionality through something called encapsulation. We give only a brief sketch here, for more details we refer the reader to the ACL2 documentation or [3]. The idea behind encapsulation is that we can define a function and prove certain properties of that function. Then, we can prove theorems about that function using only those properties (ignoring the definition of the function). Any theorems that we prove about that function then hold for any other function that also has the same properties. Consider the following example from the ACL2 documentation:

```
(encapsulate (((an-element *) => *))

    ; The list of signatures above could also be written
    ; ((an-element (lst) t))

    (local (defun an-element (lst)
        (if (consp lst)
            (car lst)
            nil)))
    (local (defthm member-equal-car
        (implies (and lst (true-listp lst))
            (member-equal (car lst) lst))))
    (defthm thm1
         (implies (null lst) (null (an-element lst))))
    (defthm thm2
         (implies (and (true-listp lst)
                (not (null lst)))
         (member-equal (an-element lst) lst))))
```

The function introduced by the encapsulate is `an-element`. The properties proven are that `(an-element lst)` is the empty list if `lst` is the empty list, and that if `lst` is a true-list and non-empty then `(an-element lst)` is contained in `lst`. Note that no property requires that the item returned be the first item, nor is it specified how the function must behave when it is given a non-list. The local definition of `an-element` is a witness showing that such a function exists.

2 ACL2 Model

2.1 Program Structure

As described earlier, LabVIEW programs essentially consist of functional nodes passing data around on wires with a semi-linear control flow. Since ACL2 is a functional language, there is a very straightforward mapping from LabVIEW programs to ACL2 where we model LabVIEW nodes with ACL2 functions that pass around associative arrays (records [2]) and wires as ACL2 functions that extract the relevant entries from the records. In other words, there is a 1-1 correspondence between LabVIEW nodes and wires, and the ACL2 functions that model them.

For example, we translate the diagram in Figure 2 into three ACL2 functions, shown in Figure 3. The important features to note are that there is one ACL2 function for each wire or node, each node takes an input record ("IN") and returns a record binding its output to a terminal name, and that wire functions take an input record and return an entry from that record. The node functions take the input record, extract their inputs by calling input wires, call the appropriate implementing function and then bind the return value from the implementing function to the appropriate terminal name. Note that the code for each node is essentially skeleton code that encapsulates the structure and control flow of the LabVIEW programs; the actual semantics come from another function, shared across nodes of the same type. The separation of structure and semantics makes it very easy to change the definition of a LabVIEW primitive without affecting the translation process or previously generated ACL2 models. The abstraction is also important for proof control in the verification process where controlling the available facts about LabVIEW primitives is fundamental to robust proofs. In addition, the separation leaves open the future possibility of multiple models of LabVIEW where the end user can choose the model he wishes to verify against without having to regenerate the program for the specific model.

2.2 Naming

Because any realistic verification requires significant interaction with the theorem prover, human readable names for the generated code is essential. Unfortunately, in LabVIEW there is no way to name wires and it is very unusual to name individual function nodes. Thus requiring user supplied names for each node would lead to an overly complicated diagram and add a significant burden to producing verifiable programs. Ergo, it is essential that the default names are useful, but at the same time users be given the possibility of controlling the naming should they choose to do so.

To this end, our generated code uses a readable default naming convention, described below, and also allows the user to select custom names for individual nodes (modulo slight modifications to maintain uniqueness). Node functions are translated to a name in the style of `type`-n, where `type` is the kind of node (e.g. `add` for an addition node) and n is a number assigned to it according to the number of other nodes of the same type we have seen before. That is, n is a number between 0 and the number of nodes of the same type on the diagram. These numbers should be regarded as non-deterministic and thus should not be relied upon to remain stable. If any theorems specifying individual nodes by name (besides auto-generated theorems) are needed to complete the verification then those nodes should be given non-default names to guarantee they remain stable in the future.

2.3 Semantic Model

Each LabVIEW primitive is modeled with an ACL2 function whose semantics are supposed to match those of the original LabVIEW as closely as possible. Since LabVIEW lacks a formal semantics, or even a single specification document, the actual semantics were determined by a combination of test executions, reference to the available documentation and conversations with LabVIEW engineers.

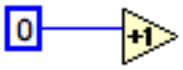

Figure 2: Simple LabVIEW Diagram

```
(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: ACL2 Model of Figure 2

LabVIEW Datatype	**ACL2 Datatype**
(8,16,32,64)-bit unsigned integer	natp
(8,16,32,64)-bit signed integer	integerp
floating point	rationalp [1]
array	true-listp [2]
string	stringp

Table 1: Table of LabVIEW datatypes in ACL2

2.3.1 LabVIEW Datatypes

Datatypes are the only place where we have made a conscious divergence from the actual LabVIEW semantics. LabVIEW integers, like those of most programming languages, are finite and distinguished by bound and signedness. But, we model all LabVIEW numerical types with ACL2's infinite idealized integers. Thus the theorems we prove are not actually theorems about LabVIEW, but rather theorems about an idealized form of LabVIEW. For example, consider Figure 4. In ACL2, `x + 1 > x` is a theorem for all `x`. But in LabVIEW, this is not true if `x` is an unsigned integer and equal to $2^{32} - 1$, for example.

This discrepancy may, quite legitimately, be considered a serious deficiency in our approach: after all, our end goal is to model LabVIEW, not idealized LabVIEW. However, for this prototype, the emphasis was on LabVIEW structural elements and the translation framework itself. This deliberate decision was made in light of two important considerations. First, the problem of reasoning about bounded arithmetic in not new and has been handled several times before in the ACL2 community. Second, once the basics of our approach

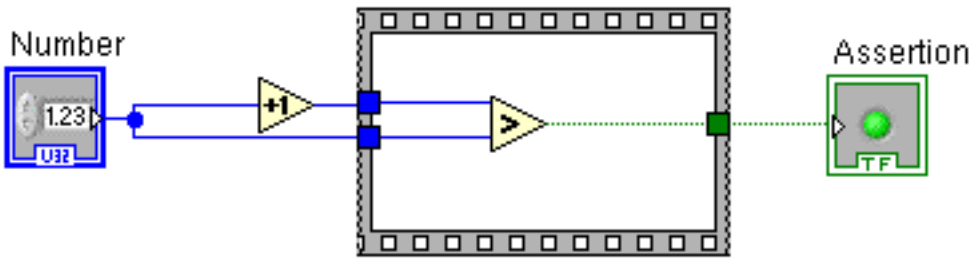

Figure 4: This is a theorem in our model, but not in LabVIEW

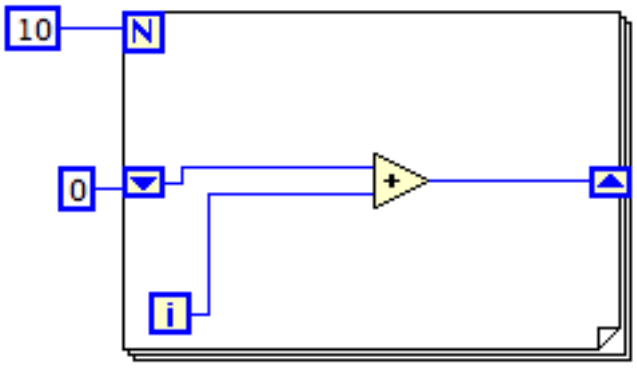

Figure 5: LabVIEW diagram with a for-loop

have been well polished, modifying the semantics to use bounded arithmetic (and utilizing the appropriate libraries for verification) should be a straight-forward proposition. Thus we feel that this decision was the correct one when our system is considered in light of both its nature as a prototype and the wider context of the work of the ACL2 community.

2.4 Model of LabVIEW Control Structures

LabVIEW's linear, static control flow can be modified with various control structures. Control structures are large blocks that can be placed on diagrams as shown in Figure 5. For more details see Section 1.3. In our current version of the translator we only translate for-loops and while-loops. These structures provide a basic and full-featured set of LabVIEW control structures and enable us to elegantly and simply construct a large number of algorithms. We illustrate a model of a for-loop below; while-loops are modeled similarly, with obvious changes to the termination tests.

The key to understanding our model of for-loops is the observation that any LabVIEW for-loop can be separated into several distinct parts. The first is the initialization of loop variables. Second is the termination test. Thirdly we have the function that represents a single step of the loop. Next we have the step that propagates the output values of the previous iteration into the input values of the next iteration. Finally we have the step that takes the output values of the final iteration and binds it to the loops output terminals.

Our model of LabVIEW for-loops splits each of these distinct parts into separate functions. The functions for the VI in Figure 5 are shown in Figure 6.

2.5 Modular Verification

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

```
(DEFUN-N |_N_5| (IN)
   (S* :|_T_1| (ADD-0<X+Y> IN)))

 (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 6: ACL2 translation of the loop structure in Figure 5

3 Translator

Now that the ACL2 model of LabVIEW has been described, I present the system that translates LabVIEW diagrams into ACL2.

3.1 IGL

The first step in the process is to convert the LabVIEW diagram from its binary format into a textual representation so that we can more easily parse and analyze it. This step is performed by an internal NI research project called the GCompiler. We don't describe its operation here; rather we just describe its output, an s-expression format call the Intermediate G Language (IGL). IGL is a "raw" representation of the LabVIEW diagram that contains virtually all of the information in the diagram. Unfortunately, IGL is not legal ACL2 due to the presence of floating point constants and illegal symbols. Thus, IGL is unsuitable for an intermediate format for the translator, which is written in ACL2.

3.2 IGML

The next step in translation is to parse the IGL into an ACL2 compatible format called Intermediate G Markup Language (IGML). IGML is quite similar in syntax to IGL, but has a more regular structure and contains only data relevant to the translation process. Irrelevant data removed from the IGML generation stage includes node positional information and wire "joints" (graphical descriptions of the bends and lengths of wires). IGML also renames nominal constants from such as `igl-node` as shown in Figure 7 to remove direct dependence upon the output of the GCompiler.

3.3 Typed IGML

Once the IGML representation has been constructed, it is immediately transformed into a typed data structure with abstract datatypes using the 'deflist' and 'defstructure' [1] ACL2 books. From this point on all code in the translator has verified guards (for details on what this means, see section 1.4). This typed format forms the intermediate representation for the rest of the translator, including code and theorem generation.

3.4 ACL2 Generation

The ACL2 code generation follows a relatively straight-forward algorithm.

1. First the typed IGML structure for the entire diagram is traversed and a data-flow dependency is constructed for the nodes. For example, if node `increment`-1 takes the output of `add`-0 as an input, then `increment`-1 depends upon `add`-0. It is necessary to determine this ordering since ACL2 requires that functions be defined in order of their use. Thus our ACL2 functions must be generated with respect to the data-flow ordering or else we won't generate legal ACL2.

2. Finally, the actual code generation occurs. The algorithm is implemented as a function which recurses down the typed IGML data structure in the ordering extracted above. At each node we call a special code generation function depending on the type of node. For example, if we encounter a function node then we call a function which looks up the specific function in a table describing the inputs, outputs and semantic function. Or, should we encounter a node representing a for-loop then we would construct the special functions described in section 2.4. One interesting property of our code generation process is that we prove our algorithm outputs a `pseudo-term-listp`. That is, our generated code is proven to be, in a weak sense, syntactically legal ACL2 code.

4 Specifications

So far I have described our model and mechanism for translating LabVIEW into ACL2, but the whole *raison d'etre* for the project, verification, requires properties to be verified. In this section I introduce our method for specifying properties to be verified.

Verification properties are specified in LabVIEW through special blocks of LabVIEW code with a single boolean output (see Figure 4). Essentially, these are similar in spirit to "assert" statements in other languages; the idea is to write some LabVIEW code that evaluates to true when the program is behaving "correctly", and false when it is misbehaving. As an example, if the diagram was intended to compute the square root of its input, then an appropriate assertion block would square the output and compare it to the input, the output of the equality test forming the boolean output of the block.

In the implementation we make a (somewhat unnecessary) distinction between "straight-line" assertions and assertions attached to loops. Straight-line assertions consist of a single block, as described above, whose only preconditions are the type assumptions on the inputs to the diagram. Loop assertions are two separate assertion blocks; one block outside the loop called the "top" assertion and one block inside the loop called the "invariant" block. The top assertion is essentially identical to a straight-line assertion and is in fact treated almost exactly the same, with the caveat that the proof of a top assertion is delayed until the proof of the associated invariant. The invariant block is an inductive invariant for the loop strong enough to prove the top invariant. That is, the invariant is a property that should hold on entry to the loop, is maintained at every step of the loop, and logically implies the top assertion at the end of the loop's execution. For more details, see Section 5. For-loops with a fixed number of iterations may not require an invariant and thus may have a trivial invariant block (a block that outputs the constant true).

5 Verification Procedure

I present our verification methodology through an example. I begin by explaining the example in complete detail. Then I explain the structure of the generated theorems for verification.

Note that this proof structure was originally devised and worked by hand by Matt Kaufmann for a specific example. It has changed slightly from working many examples and being auto-generated, but it is still primarily Matt's contribution.

5.1 Main Diagram Contents

We describe the diagram in Figure 8 from left to right, top to bottom. The leftmost item, labeled "Array" is the initial input to the diagram, representing the array to be sorted. Note that at the bottom of the node is "I 32", symbolizing that this is an array of unsigned 32 bit integers. Since LabVIEW currently lacks proper polymorphism, our diagram is directly tied to this type. However our proofs make no use of this type and in fact work when the type is changed to arbitrary numeric LabVIEW types. Next is an array length node that outputs the length of the input array, which is in turn used as the bound for the for-loop. Below this is a node consisting of two blocks with '0' written inside of them. This represents constant empty 1-D array that is used to initialize the output array.

Next we have the for-loop structure itself in light blue. At the top of the loop is a block with a capital 'N' inside which represents the loop bound. This is set upon entry to the loop and is not changed once the loop is started. In this case it is set to the length of the input array. Next on the border of the loop is a shift register that stores the sorted array that is built up at each iteration of the loop. On entry to the loop it is initialized to the wired value given by the constant array node described above. At each step its new value is the value stored in its sibling register (directly opposite it on the loop border) in the previous iteration. Next is a blue block representing a constant loop input. At every iteration its value is the original input to the loop, in this case the input array to the diagram. The blue-outlined white box with an 'i' in it is a variable that represents the current loop counter at each iteration of the loop. Note that the first iteration is '0' and the last value of the loop counter that is seen inside the loop is N - 1. Leaving the yellow block for later, the next element we encounter is an array indexing node. This particular node returns the input array element stored at index i where i is the current value of the loop counter. It then feeds this value to the "insert" subVI, which also takes the current version of the sorted array we are building up. The "insert" subVI returns a new sorted array containing the old array and the item to be inserted. This is in turn fed into the shift register for the next iteration. Here we use our modular verification procedure mentioned in 2.5. The exact properties exported are shown in Appendix E.

Once the loop has run to completion, the last value of the shift register is sent on the output wire which is fed to both the green assertion block and the output element "Array 2".

The large yellow block inside the for-loop is an assertion block, specifically a loop invariant block, which is distinguished from other kinds of assertion blocks by its color. Regular LabVIEW users will recognize this as a flat sequence block. Since our subset of LabVIEW is purely functional, forcing code to execute in a particular sequence has no effect on the semantics so we have adopted these blocks for the sole purpose of embedding assertions. The blue block on the left side of the assertion block is an input to the block similar to the loop input. In this case it can be ignored and treated as if a single wire continued on uninterrupted by the input. The pink-outlined box with "sortedp" represents a string constant. The white block with "ACL2 Pred 1 Arg" represents a "cosmetic" node. Nominally this is a subVI call, but we translate it specially. Instead of using an actual subVI we instead replace the call with an ACL2 function whose name is given by the first argument ("sortedp") and whose first parameter is given by the second node argument. In other words, the semantic value of this node here is given by the value of ACL2's "sortedp" on the array representing the sorted array we are building. The white block "Loop 1" identifies the assertion and the loop it is attached to. In this case it is clearly superfluous since the assertion block is clearly contained in the loop, but this was not the case in previous versions of the translator and has been left as a vestigial artifact. Next we have a boolean (green) output of the assertion block which is fed into an output element labeled "Loop Inv".

```
(igl_node   _n_7
    (class Function)
    (name Array%20Size)
    (rect 35_202_67_223)
    (diagram _n_0)
    (terminals  (
        (_t_0  (
            (type (LVArray (-1) LVInt32_Numeric))
            (link (_n_1  _t_1))
            (termtype SimpleIn)
            (tname array)
        ))
        (_t_1  (
            (type LVInt32)
            (link (_n_4  _t_0))
            (termtype SimpleOut)
            (tname size%28s%29)
        ))
    ))
)
```

Figure 7: IGL example

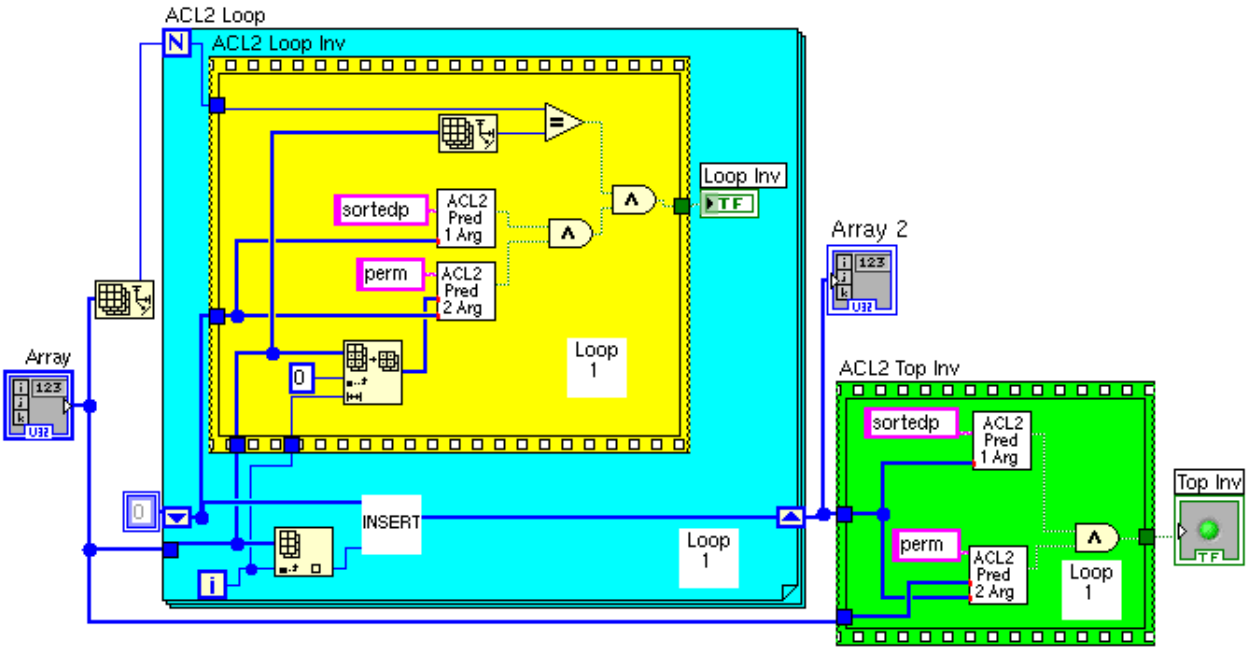

Figure 8: Insertion Sort diagram

```
(IN-PACKAGE "ACL2")

(INCLUDE-BOOK "insertion-sort-fns")

(LOCAL (INCLUDE-BOOK "insertion-sort-work"))

(SET-ENFORCE-REDUNDANCY T)

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

Figure 9: insertion-sort.lisp

The green block is another assertion block, representing a loop assertion. In this case its contents are exactly the same as the yellow invariant block described above.

5.2 Insertion Diagram Output

The translator generates 3 separate files for a single diagram. Each of these files is intended to be certified. For the complete content of the actual files for the insertion-sort example, please see Appendices B,C,D. The naming convention is standard, substitute for "insertion-sort" to find the proper filenames for other diagrams.

The first file generated is the functions book (insertion-sort-fns.lisp) which contains the ACL2 model of the diagram as described in Section 2.

The top book (insertion-sort.lisp), shown in Figure 9, includes the functions file and contains the statement of the "top" assertion for the loop. In general, this file contains all of the "top" and "inline" assertions for a specific diagram, it doesn't contain any loop invariants or proof scaffolding. In this file, `ACL2-TOP-INV` is the name of the "top" assertion block, `:ASN` is the (boolean) output key of that block, and `INSERTION-SORT$INPUT-HYPS` is a predicate stating the type hypotheses on the diagram inputs.

The work file (insertion-sort-work.lisp) is locally included by the top book. Because the top book locally includes the work file, the main theorem (`ACL2-TOP-INV-$INV`) is guaranteed to be a logical consequence of the function definitions in the fns file [3]. The intention here is that the work file is the users scratch pad, and the actual validity of the verification relies solely upon the successful certification of the top file. This allows the user to modify the work file to her heart's content, confident that, as long as the top file is not modified, the various machinations can't compromise the verification process.

The primary interesting aspect of the work file is that this is where all of our theorem generation output goes. The goal is to automate the construction of the work file as much as possible so that only a minimum of actual user work is required to complete the verification process. In the example shown here, no user work is required to complete the verification. We carefully control the verification process by generating a highly structured proof scaffolding. To provide as much control of the proof process as possible, we employ a generic theory which contains the basic structure of inductive invariants. To wit, we provide a generic theory that defines an encapsulated loop function and a generic loop invariant and then shows that this generic loop invariant is preserved by running the generic loop to completion. This enables us to avoid explicit induction in the proof, by using functional instantiation (see Section 1.5 for more details on encapsulation).

At a high level we split the proof into several parts. First we verify that the loop invariant is preserved by a single step, which implies that it holds until the loop runs to completion. Then we show that the invariant holds when the loop begins. At this point we have verified that the invariant is actually a loop invariant. Next, we show that the loop invariant implies the top level assertion. This structure is set into a generic

theory, where we prove that a generic invariant that is preserved for each step must in turn be preserved for the whole execution.

Now we examine the work file generated for the diagram shown in Figure 8 in detail, showing how our generated proof scaffolding structure firmly directs the proof outlined above. All of the main lemmas and theorems appear below. Some minor and uninteresting technical lemmas are left out. For the complete file, see Appendix C.

Extend the loop invariant

```
(DEFUN |_N_35$HYPS| (IN)
       (AND (NATP (G :LC IN))
            (INTEGERP (G :|_T_2| IN))
            (INTEGER-LISTP (G :|_T_4| IN))
            (INTEGER-LISTP (G :|_T_5| IN))))


(DEFUN |_N_35$PROP| (N IN)
       (DECLARE (IGNORABLE N))
       (AND (|_N_35$HYPS| IN)
            (EQUAL N (G :|_T_2| IN))
            (G :ASN (ACL2-LOOP-INV-FLATSEQUENCE_N_13 IN))))
```

Here we extend the user supplied invariant by conjoining it to the type requirements on the input to the (inner) loop (represented by `|_N_35$HYPS|`). The second conjunction says that the loop bound (`|T_2_|`) is bound to `N`.

The loop invariant is preserved by taking a step of loop

```
(DEFTHMDL |_N_35$PROP_N_10$STEP|
          (IMPLIES (AND (NATP (G :LC IN))
                        (< (G :LC IN) N)
                        (|_N_35$PROP| N IN))
                   (|_N_35$PROP| N
                                 (S :LC (1+ (G :LC IN))
                                    (|_N_10$STEP| IN)))))
```

Next we prove that the loop invariant is preserved after taking a step of the loop. This corresponds to the inductive step of the proof. In general, this may require additional diagram specific lemmas, but in our example it proves automatically as generated.

The loop invariant is preserved by running the (inner) loop to completion

This step says that the invariant is preserved until completion of the loop. In order to prove it, we use the generic theory mentioned above. This allows us to avoid explicit induction and, in fact, the proof should be completely automatic. Note the use of `functional-instance`. This is where we explicitly tell ACL2 to use the generic theory, and in fact we disable all but a minimal number of other theorems to strictly direct the proof process.

```
(DEFTHML
  |_N_35$PROP_N_10|
  (IMPLIES (AND (NATP N)
                (NATP (G :LC IN))
```

```
              (|_N_35$PROP| N IN))
          (|_N_35$PROP| N (|_N_10$LOOP| N IN)))
  :HINTS
  (("Goal"
        :BY (:FUNCTIONAL-INSTANCE LOOP-GENERIC-THM
                                  (KEYS-GENERIC |_N_10$KEYS|)
                                  (STEP-GENERIC |_N_10$STEP|)
                                  (PROP-GENERIC |_N_35$PROP|)
                                  (LOOP-GENERIC |_N_10$LOOP|))
        :IN-THEORY
        (UNION-THEORIES '(|_N_35$PROP_N_10$STEP| |_N_10$STEP-S| (|_N_10$KEYS|)
                                                 (MEMBER))
                        (THEORY 'MINIMAL-THEORY))
        :EXPAND ((|_N_10$LOOP| N IN))))
  :RULE-CLASSES NIL)
```

The loop invariant holds for the outer loop

```
(DEFUN ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPRE
       (IN)
       (AND (INSERTION-SORT$INPUT-HYPS IN)
            (NATP (|ARRAY-SIZE-0<SIZE(S)>| IN))))

(DEFTHML ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVINIT
         (IMPLIES (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPRE IN)
                  (|_N_35$PROP| (|ARRAY-SIZE-0<SIZE(S)>| IN)
                                (|_N_35$PROP$INIT| IN)))
         :RULE-CLASSES NIL)
```

Here we show that the loop invariant holds on entry to the loop. The first definition here is a predicate for the input hypotheses to the whole diagram. |ARRAY-SIZE-0<SIZE(S)>| represents the loop bound (the size of the array to be sorted). The theorem should be read as stating that if the inputs to the diagram are properly typed, then the loop invariant is true of the inputs to the loop.

```
(DEFUN ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV+
       (IN)
       (DECLARE (XARGS :NORMALIZE NIL))
       (|_N_35$PROP| (|ARRAY-SIZE-0<SIZE(S)>| IN)
                     (|_N_10$LOOP| (|ARRAY-SIZE-0<SIZE(S)>| IN)
                                   (|_N_10$LOOP$INIT| IN))))

(DEFTHML
     ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV$CONDITIONAL
     (IMPLIES (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPRE IN)
              (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV+ IN))
     :HINTS (("Goal" :USE ((:INSTANCE |_N_35$PROP_N_10|
                                      (IN (|_N_35$PROP$INIT| IN))
                                      (N (|ARRAY-SIZE-0<SIZE(S)>| IN)))
                           ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVNATP-N
                           ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVINIT
                           ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVNATP-LOOP-INIT
                           ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV+)
                     :IN-THEORY (UNION-THEORIES '((NATP) |_N_35$PROP$INIT|)
                                                (THEORY 'MINIMAL-THEORY)))))
```

```
 :RULE-CLASSES NIL)
```

Now we show that the loop invariant holds when we actually run the loop. The predicate simply says that the loop invariant holds when the loop is run from start to finish, and the theorem states that if the invariant holds on entrance then it holds when the loop ends.

```
(DEFTHML
 ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV
 (IMPLIES (INSERTION-SORT$INPUT-HYPS IN)
          (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV+ IN))
 :HINTS
 (("Goal" :IN-THEORY (UNION-THEORIES '(ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPRE)
                                     (THEORY 'MINIMAL-THEORY))
          :USE (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV$CONDITIONAL
                    ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPREHOLDS)))
 :RULE-CLASSES NIL)
```

Now we put the two previous theorems and together and show that the invariant holds for the loop, assuming only the type hypotheses on the diagram. To recap, we first showed that the invariant held on entry to the loop, then showed that if it held on entry then it held on exit, and then stitched the two facts together (essentially modus ponens) to conclude that the loop invariant held on exit to the loop.

The top assertion is true

```
(DEFTHM
 ACL2-TOP-INV$INV
 (IMPLIES (INSERTION-SORT$INPUT-HYPS IN)
          (G :ASN (ACL2-TOP-INV IN)))
 :HINTS (("Goal" :IN-THEORY (DISABLE |_N_10$LOOP|)
                 :USE (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV LEMMA-2-ACL2-LOOP))))
```

Now we show that the top level assertion holds. In general we can't really give much help with this proof. If the loop invariant is chosen carefully then this theorem should simply fall out directly, but it is reasonable to expect that this will take some work. The theorem here simply states that the top invariant holds if the inputs to the diagram are properly typed.

6 Theorem Library

One very important part of a verification framework for a language is an expansive set of existing 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.

Because we choose to translate LabVIEW into ACL2 and prove theorems about ACL2 implementations of LabVIEW primitives, it could be argued that we really aren't proving anything about LabVIEW programs but rather proving things about the nitty-gritty details of our specific ACL2 implementation. It is also 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 are true of the actual LabVIEW primitives instead of the specific ACL2 implementation of the primitives which may have

additional properties. At the current stage of development we simply leave the definitions disabled, but in the future once the library has stabilized we may wish to replace the actual functions with an encapsulate which captures the essential properties of each primitive.

7 Conclusion

In this thesis I have presented a methodology for modeling, annotating and verifying LabVIEW programs as well as an implementation of this methodology. I have developed an extensive library of both ACL2 functions modeling LabVIEW primitives and of theorems about these primitives. With the system and the library we have annotated and verified[1] more than a dozen LabVIEW programs, two of which I have presented here as examples.

7.1 Soundness

I view the soundness of our results in two separate categories. There is first the soundness of verification with respect to the original LabVIEW program. Then there is the soundness of our verification with respect to the ACL2 model of the LabVIEW program. It is clear that this second soundness condition depends solely upon the soundness of ACL2, so I leave it at that and instead focus on the former category.

Soundness with respect to LabVIEW is, at the moment, impossible to formalize as LabVIEW lacks any formal semantics. As such, the closest thing we have is a reference implementation. Obviously, this is not a useful model to reason with. However, in our work we have established what we think should be an appropriate sematics of LabVIEW (modulo some important details that will be mentioned later) by careful reading of the documentation and simulation. We have primarily foused on a relatively simple but fully featured subset of LabVIEW and we believe that our current semantics agrees with the available implementations of LabVIEW. The caveat mentioned earlier is the use of bounded arithmetic, floating point numbers and bounded data structures. LabVIEW uses standard bound numeric types such as 32-bit or 64-bit integers, but our current model only uses ACL2's idealized integer arithmetic. Thus our semantics will likely disagree with LabVIEW in cases of overflow. This choice was made primarily out of pragmatic concerns as the expense of dealing with modular arithmetic was deemed excessive compared to the benefit of a more accurate model.

The other part that has to be relied upon is our actual translation. The argument here is that the structural match between ACL2 and LabVIEW is quite close, thus the translation is correspondingly simple and the likelihood of errors undetected by our testing is very low. Furthermore, in the absence of a semantics for the source language, it isn't possible to do any better.

References

[1] Bishop Brock. Defstructure for ACL2. Technical report, Computational Logic, Inc, 1997.

[2] 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.

[3] Matt Kaufmann and J. Strother Moore. Structured theory development for a mechanized logic. *J. Autom. Reason.*, 26(2):161–203, 2001.

[4] J Strother Moore Matt Kaufmann, Panagatios Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[1] See section 7.1 for exactly what it means to be verified

A Introduction to ACL2

ACL2, roughly speaking, is a unique combination of a programming language, a computational logic and an automated theorem prover. The purpose of this appendix is to provide the reader a background in ACL2 sufficient to understand all of the elements in this thesis. If the reader wishes to gain an effective understanding of ACL2, we point them to the book by Kaufmann, Manolios and Moore [4].

A.1 ACL2 the Language

ACL2 is essentially a total, first-order, applicative (purely functional) subset of Common Lisp. Total means that all functions are defined on all possible inputs. First order means that functions can not be used as arguments to other functions. Applicative means that functions can not hold state, that is, a function is truly a function of its inputs. More concretely, if you have a function named `f` that takes one argument, then `(f 1)` will always return the same value, regardless of what has happened before `f` was called.

Syntactically, ACL2 uses the (in)famous Lisp S-Expression (Symbolic Expression) syntax. The defining syntactic elements of S-expressions are the use of balanced parentheses along with a prefix notation. Whereas in a C/Java style language you might use a combination of brackets, parentheses and braces to denote the scope of certain expression, in ACL2 we use only the parentheses. This is not as awkward as one might expect since in ACL2 everything is (roughly speaking) a function application. To demonstrate exactly what is meant by this, we walk you through the definition of a simple ACL2 function.

Consider the code shown in Figure 11.

A.2 ACL2 the Logic

A.3 ACL2 the Theorem Prover

B Insertion Sort Functions File

```
(IN-PACKAGE "ACL2")

(LOGIC)

(SET-IGNORE-OK T)

(SET-IRRELEVANT-FORMALS-OK T)

(DEFLABEL START-INSERTION-SORT)

(DEFUN-N CONSTANT[NIL]-0 (IN)
         (S* :|_T_0| 'NIL))

(DEFUN-W CONSTANT[NIL]-0<_T_0> (IN)
         (G :|_T_0| (CONSTANT[NIL]-0 IN)))

(DEFUN-W ARRAY<ARRAY> (IN)
         (G :ARRAY IN))

(DEFUN-N ARRAY-SIZE-0 (IN)
         (S* :|SIZE(S)| (LEN (ARRAY<ARRAY> IN))))

(DEFUN-W |ARRAY-SIZE-0<SIZE(S)>| (IN)
         (G :|SIZE(S)| (ARRAY-SIZE-0 IN)))

(DEFUN-W |_N_10<_T_2>| (IN)
```

```
        (G :|_T_2| IN))

(DEFUN-W |_N_10<_T_5>| (IN)
        (G :|_T_5| IN))

(DEFUN-W |_N_10<_T_4>| (IN)
        (G :|_T_4| IN))

(DEFUN-W |_N_10<_T_3>| (IN) (G :LC IN))

(DEFUN-N INDEX-ARRAY-0 (IN)
        (S* :ELEMENT (INDEX-ARRAY (|_N_10<_T_3>| IN)
                                  (|_N_10<_T_4>| IN)
                                  '0)))

(DEFUN-W |_N_35<_T_4>| (IN)
        (G :|_T_4| IN))

(DEFUN-W |_N_35<_T_3>| (IN)
        (G :|_T_3| IN))

(DEFUN-W |_N_35<_T_1>| (IN)
        (G :|_T_1| IN))

(DEFUN-W |_N_35<_T_2>| (IN)
        (G :|_T_2| IN))

(DEFUN-N CONSTANT[0]-1 (IN)
        (S* :|_T_0| 0))

(DEFUN-W CONSTANT[0]-1<_T_0> (IN)
        (G :|_T_0| (CONSTANT[0]-1 IN)))

(DEFUN-N CONSTANT[PERM]-2 (IN)
        (S* :|_T_0| 'PERM))

(DEFUN-W CONSTANT[PERM]-2<_T_0> (IN)
        (G :|_T_0| (CONSTANT[PERM]-2 IN)))

(DEFUN-N CONSTANT[SORTEDP]-3 (IN)
        (S* :|_T_0| 'SORTEDP))

(DEFUN-W CONSTANT[SORTEDP]-3<_T_0> (IN)
        (G :|_T_0| (CONSTANT[SORTEDP]-3 IN)))

(DEFUN-N ARRAY-SIZE-1 (IN)
        (S* :|SIZE(S)| (LEN (|_N_35<_T_3>| IN))))

(DEFUN-N ARRAY-SUBSET-0 (IN)
        (S* :SUBARRAY (ARRAY-SUBSET (|_N_35<_T_2>| IN)
                                    (CONSTANT[0]-1<_T_0> IN)
                                    (|_N_35<_T_3>| IN))))

(DEFUN-N ACL2-FN-1-ARG.VI-0 (IN)
        (S* :BOOLEAN (SORTEDP (|_N_35<_T_1>| IN))))
```

```
(DEFUN-W |ARRAY-SIZE-1<SIZE(S)>| (IN)
         (G :|SIZE(S)| (ARRAY-SIZE-1 IN)))

(DEFUN-W ARRAY-SUBSET-0<SUBARRAY> (IN)
         (G :SUBARRAY (ARRAY-SUBSET-0 IN)))

(DEFUN-W ACL2-FN-1-ARG.VI-0<BOOLEAN> (IN)
         (G :BOOLEAN (ACL2-FN-1-ARG.VI-0 IN)))

(DEFUN-N EQUAL?-0 (IN)
         (S* :X-=-Y? (EQUAL (|_N_35<_T_4>| IN)
                            (|ARRAY-SIZE-1<SIZE(S)>| IN))))

(DEFUN-N ACL2-FN-2-ARG.VI-0 (IN)
         (S* :BOOLEAN (PERM (ARRAY-SUBSET-0<SUBARRAY> IN)
                            (|_N_35<_T_1>| IN))))

(DEFUN-W EQUAL?-0<X-=-Y?> (IN)
         (G :X-=-Y? (EQUAL?-0 IN)))

(DEFUN-W ACL2-FN-2-ARG.VI-0<BOOLEAN> (IN)
         (G :BOOLEAN (ACL2-FN-2-ARG.VI-0 IN)))

(DEFUN-N AND-0 (IN)
         (S* :X-.AND.-Y? (AND (ACL2-FN-1-ARG.VI-0<BOOLEAN> IN)
                              (ACL2-FN-2-ARG.VI-0<BOOLEAN> IN))))

(DEFUN-W AND-0<X-.AND.-Y?> (IN)
         (G :X-.AND.-Y? (AND-0 IN)))

(DEFUN-N AND-1 (IN)
         (S* :X-.AND.-Y? (AND (EQUAL?-0<X-=-Y?> IN)
                              (AND-0<X-.AND.-Y?> IN))))

(DEFUN-W AND-1<X-.AND.-Y?> (IN)
         (G :X-.AND.-Y? (AND-1 IN)))

(DEFUN-N |_N_35| (IN)
         (S* :ASN (AND-1<X-.AND.-Y?> IN)))

(DEFUN |_N_35$INIT| (IN)
       (S* :|_T_1| (|_N_10<_T_5>| IN)
           :|_T_2| (|_N_10<_T_3>| IN)
           :|_T_3| (|_N_10<_T_4>| IN)
           :|_T_4| (|_N_10<_T_2>| IN)))

(DEFUN-ASN ACL2-LOOP-INV-FLATSEQUENCE_N_13 (IN)
           (|_N_35| (|_N_35$INIT| IN)))

(DEFUN-W INDEX-ARRAY-0<ELEMENT> (IN)
         (G :ELEMENT (INDEX-ARRAY-0 IN)))

(INCLUDE-BOOK "insert.vi")

(DEFUN-N INSERT.VI-0 (IN)
         (S* :ARRAY-2 (INSERT.VI (|_N_10<_T_5>| IN)
```

```
                                (INDEX-ARRAY-0<ELEMENT> IN))))

(DEFUN-W INSERT.VI-0<ARRAY-2> (IN)
         (G :ARRAY-2 (INSERT.VI-0 IN)))

(DEFUN-N |_N_10| (IN)
         (S* :|_T_1| (INSERT.VI-0<ARRAY-2> IN)))

(DEFUN |_N_10$STEP| (IN)
       (S :|_T_5| (G :|_T_1| (|_N_10| IN)) IN))

(DEFUN |_N_10$LOOP| (N IN)
       (DECLARE (XARGS :MEASURE (NFIX (- N (G :LC IN)))))
       (COND ((OR (>= (G :LC IN) N)
                  (NOT (NATP N))
                  (NOT (NATP (G :LC IN))))
              IN)
             (T (|_N_10$LOOP| N
                              (S :LC (1+ (G :LC IN))
                                 (|_N_10$STEP| IN))))))

(DEFUN |_N_10$LOOP$INIT| (IN)
       (S* :LC 0
           :|_T_2| (|ARRAY-SIZE-0<SIZE(S)>| IN)
           :|_T_4| (ARRAY<ARRAY> IN)
           :|_T_5| (CONSTANT[NIL]-0<_T_0> IN)))

(DEFUN-N ACL2-LOOP (IN)
         (|_N_10$LOOP| (|ARRAY-SIZE-0<SIZE(S)>| IN)
                       (|_N_10$LOOP$INIT| IN)))

(DEFUN-W ACL2-LOOP<_T_5> (IN)
         (G :|_T_5| (ACL2-LOOP IN)))

(DEFUN-W |_N_8<_T_2>| (IN)
         (G :|_T_2| IN))

(DEFUN-W |_N_8<_T_1>| (IN)
         (G :|_T_1| IN))

(DEFUN-N CONSTANT[PERM]-4 (IN)
         (S* :|_T_0| 'PERM))

(DEFUN-W CONSTANT[PERM]-4<_T_0> (IN)
         (G :|_T_0| (CONSTANT[PERM]-4 IN)))

(DEFUN-N CONSTANT[SORTEDP]-5 (IN)
         (S* :|_T_0| 'SORTEDP))

(DEFUN-W CONSTANT[SORTEDP]-5<_T_0> (IN)
         (G :|_T_0| (CONSTANT[SORTEDP]-5 IN)))

(DEFUN-N ACL2-FN-2-ARG.VI-1 (IN)
         (S* :BOOLEAN (PERM (|_N_8<_T_2>| IN)
                            (|_N_8<_T_1>| IN))))
```

```
(DEFUN-N ACL2-FN-1-ARG.VI-1 (IN)
         (S* :BOOLEAN (SORTEDP (|_N_8<_T_1>| IN))))

(DEFUN-W ACL2-FN-2-ARG.VI-1<BOOLEAN> (IN)
         (G :BOOLEAN (ACL2-FN-2-ARG.VI-1 IN)))

(DEFUN-W ACL2-FN-1-ARG.VI-1<BOOLEAN> (IN)
         (G :BOOLEAN (ACL2-FN-1-ARG.VI-1 IN)))

(DEFUN-N AND-2 (IN)
         (S* :X-.AND.-Y? (AND (ACL2-FN-1-ARG.VI-1<BOOLEAN> IN)
                              (ACL2-FN-2-ARG.VI-1<BOOLEAN> IN))))

(DEFUN-W AND-2<X-.AND.-Y?> (IN)
         (G :X-.AND.-Y? (AND-2 IN)))

(DEFUN-N |_N_8| (IN)
         (S* :ASN (AND-2<X-.AND.-Y?> IN)))

(DEFUN |_N_8$INIT| (IN)
       (S* :|_T_1| (ACL2-LOOP<_T_5> IN)
           :|_T_2| (ARRAY<ARRAY> IN)))

(DEFUN-ASN ACL2-TOP-INV (IN)
           (|_N_8| (|_N_8$INIT| IN)))

(DEFUN-W ACL2-TOP-INV<_T_2> (IN)
         (G :|_T_2| (ACL2-TOP-INV IN)))

(DEFUN INSERTION-SORT$INPUT-HYPS (IN)
       (AND (INTEGER-LISTP (G :ARRAY IN))))

(DEFTHEORY INSERTION-SORT
           (SET-DIFFERENCE-THEORIES (CURRENT-THEORY :HERE)
                                    (CURRENT-THEORY 'START-INSERTION-SORT)))
```

C Insertion Sort Work File

```
(IN-PACKAGE "ACL2")

(LOGIC)

(LOCAL (INCLUDE-BOOK "generic-loop-inv"
                     :DIR :BASIS))

(LOCAL (INCLUDE-BOOK "library/top"
                     :DIR :BASIS))

; ------------------------------
; PRELIMINARY DEFINITIONS
; ------------------------------

(DEFUN |_N_35$PROP$INIT| (IN)
       (DECLARE (XARGS :NORMALIZE NIL))
```

```
       (|_N_10$LOOP$INIT| IN))

; -----------------------------
; PROOF OF LOOP INVARIANT PRESERVATION, IN ISOLATION
; -----------------------------

(DEFUN |_N_35$HYPS| (IN)
       (AND (NATP (G :LC IN))
            (INTEGERP (G :|_T_2| IN))
            (INTEGER-LISTP (G :|_T_4| IN))
            (INTEGER-LISTP (G :|_T_5| IN))))

(DEFUN |_N_10$KEYS| NIL
       '(:LC :|_T_2| :|_T_3| :|_T_4| :|_T_5|))

(DEFUN |_N_35$PROP| (N IN)
       (DECLARE (IGNORABLE N))
       (AND (|_N_35$HYPS| IN)
            (EQUAL N (G :|_T_2| IN))
            (G :ASN (ACL2-LOOP-INV-FLATSEQUENCE_N_13 IN))))

; Here is the key lemma, which in principle at least requires some work by the
; prover, and may require help from the user.  It corresponds to the inductive
; step.

; USER ASSISTANCE MAY BE REQUIRED:


(DEFTHMDL |_N_35$PROP_N_10$STEP|
          (IMPLIES (AND (NATP (G :LC IN))
                        (< (G :LC IN) N)
                        (|_N_35$PROP| N IN))
                   (|_N_35$PROP| N
                                (S :LC (1+ (G :LC IN))
                                   (|_N_10$STEP| IN)))))

(DEFTHMDL |_N_10$STEP-S|
          (IMPLIES (NOT (MEMBER KEY (|_N_10$KEYS|)))
                   (EQUAL (|_N_10$STEP| (S KEY VAL IN))
                          (S KEY VAL (|_N_10$STEP| IN)))))

(DEFTHML
  |_N_35$PROP_N_10|
  (IMPLIES (AND (NATP N)
                (NATP (G :LC IN))
                (|_N_35$PROP| N IN))
           (|_N_35$PROP| N (|_N_10$LOOP| N IN)))
  :HINTS
  (("Goal"
        :BY (:FUNCTIONAL-INSTANCE LOOP-GENERIC-THM
                                  (KEYS-GENERIC |_N_10$KEYS|)
                                  (STEP-GENERIC |_N_10$STEP|)
                                  (PROP-GENERIC |_N_35$PROP|)
                                  (LOOP-GENERIC |_N_10$LOOP|))
        :IN-THEORY
        (UNION-THEORIES '(|_N_35$PROP_N_10$STEP| |_N_10$STEP-S| (|_N_10$KEYS|)
```

```
                                                (MEMBER))
                        (THEORY 'MINIMAL-THEORY))
          :EXPAND ((|_N_10$LOOP| N IN))))
  :RULE-CLASSES NIL)

(DEFTHML LC$_N_10
         (IMPLIES (AND (NATP N)
                       (NATP (G :LC IN))
                       (<= (G :LC IN) N))
                  (EQUAL (G :LC (|_N_10$LOOP| N IN)) N))
         :HINTS (("Goal" :BY (:FUNCTIONAL-INSTANCE LOOP-GENERIC-LC
                                                   (KEYS-GENERIC |_N_10$KEYS|)
                                                   (STEP-GENERIC |_N_10$STEP|)
                                                   (PROP-GENERIC |_N_35$PROP|)
                                                   (LOOP-GENERIC |_N_10$LOOP|))
                         :IN-THEORY (THEORY 'MINIMAL-THEORY)
                         :EXPAND ((|_N_10$LOOP| N IN)))))

; ------------------------------
; CORRECTNESS OF LOOP INVARIANT
; WITH RESPECT TO THE ENVIRONMENT,
; BUT CONDITIONAL ON PRECONDITION
; ------------------------------

; We prove that the outer loop invariant block holds, under assumptions on its
; inputs, namely: type assumptions on the diagram's top-level inputs, as well
; as assumptions coming from the associated loop precondition block.  But at
; first we cheat, acting as though the outer loop invariant block calls the
; loop block after extending the loop block's input with ghost variables.  Then
; we functionally instantiate s-alist-commutes-generic to get the desired
; result.

(DEFUN ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPRE
       (IN)
       (AND (INSERTION-SORT$INPUT-HYPS IN)
            (NATP (|ARRAY-SIZE-0<SIZE(S)>| IN))))

(DEFTHML ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVNATP-N
         (IMPLIES (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPRE IN)
                  (NATP (|ARRAY-SIZE-0<SIZE(S)>| IN)))
         :HINTS (("Goal" :EXPAND ((ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPRE IN))
                         :IN-THEORY (THEORY 'MINIMAL-THEORY)))
         :RULE-CLASSES NIL)

; USER ASSISTANCE MAY BE REQUIRED:
; This is a real proof obligation!  It implements the requirement that the
; invariant hold when initially entering the loop, and is thus probably trivial
; to prove in most cases.

; If a non-trivial precondition block has been specified for the loop
; invariant, then the invariant hypotheses may be enough to prove the property
; without opening up any signals in the actual design.  So you may wish to
; disable the inputs to the loop invariant block, from _N_xx$LOOP$INIT.  While
; that approach may speed up the proof of this lemma, it could make the proof
; fail.
```

```
(DEFTHML ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVINIT
         (IMPLIES (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPRE IN)
                  (|_N_35$PROP| (|ARRAY-SIZE-0<SIZE(S)>| IN)
                                (|_N_35$PROP$INIT| IN)))
         :RULE-CLASSES NIL)

(DEFTHMDL
  ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVNATP-LOOP-INIT
  (EQUAL (G :LC (|_N_35$PROP$INIT| IN)) 0)
  :HINTS
  (("Goal" :EXPAND ((|_N_35$PROP$INIT| IN)
                    (|_N_10$LOOP$INIT| IN))
           :IN-THEORY (UNION-THEORIES '(S-ALIST)
                                      (CURRENT-THEORY 'START-INSERTION-SORT)))))

(DEFUN ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV+
       (IN)
       (DECLARE (XARGS :NORMALIZE NIL))
       (|_N_35$PROP| (|ARRAY-SIZE-0<SIZE(S)>| IN)
                     (|_N_10$LOOP| (|ARRAY-SIZE-0<SIZE(S)>| IN)
                                   (|_N_10$LOOP$INIT| IN))))

(DEFTHML
     ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV$CONDITIONAL
     (IMPLIES (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPRE IN)
              (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV+ IN))
     :HINTS (("Goal" :USE ((:INSTANCE |_N_35$PROP_N_10|
                                      (IN (|_N_35$PROP$INIT| IN))
                                      (N (|ARRAY-SIZE-0<SIZE(S)>| IN)))
                           ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVNATP-N
                           ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVINIT
                           ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVNATP-LOOP-INIT
                           ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV+)
                     :IN-THEORY (UNION-THEORIES '((NATP) |_N_35$PROP$INIT|)
                                                (THEORY 'MINIMAL-THEORY))))
     :RULE-CLASSES NIL)

; ------------------------------
; CORRECTNESS OF LOOP INVARIANT
; WITH RESPECT TO THE ENVIRONMENT
; ------------------------------

; First we prove the precondition.  In general, this could take some work.

; USER ASSISTANCE MAY BE REQUIRED:

(DEFTHML ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPREHOLDS
         (IMPLIES (INSERTION-SORT$INPUT-HYPS IN)
                  (AND (NATP (|ARRAY-SIZE-0<SIZE(S)>| IN))))
         :RULE-CLASSES NIL)

; Now we use the conditional invariant result to prove the desired assertion.

(DEFTHML
 ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV
 (IMPLIES (INSERTION-SORT$INPUT-HYPS IN)
```

```
         (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV+ IN))
 :HINTS
 (("Goal" :IN-THEORY (UNION-THEORIES '(ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPRE)
                                     (THEORY 'MINIMAL-THEORY))
          :USE (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV$CONDITIONAL
                    ACL2-LOOP-INV-FLATSEQUENCE_N_13$INVPREHOLDS)))
 :RULE-CLASSES NIL)

; ------------------------------
; CORRECTNESS OF MAIN ASSERTION BLOCK
; ------------------------------

; Our goal is to derive the main assertion as a corollary of the loop
; invariant.  Thus, we do not want to open up the self-reference (inner) node
; of the loop block.  The following lemmas extract what we
; need from that loop block.

(DEFTHML
  LEMMA-1-ACL2-LOOP
  (IMPLIES (NOT (MEMBER-EQ KEY (CONS :LC '(:|_T_5|))))
           (EQUAL (G KEY (|_N_10$LOOP| N IN))
                  (G KEY IN)))
  :HINTS
  (("Goal" :IN-THEORY (UNION-THEORIES '(|_N_10$LOOP| |_N_10$STEP|)
                                      (CURRENT-THEORY 'START-INSERTION-SORT)))))

(DEFTHML LEMMA-2-ACL2-LOOP
         (IMPLIES (INSERTION-SORT$INPUT-HYPS IN)
                  (NATP (|ARRAY-SIZE-0<SIZE(S)>| IN)))
         :HINTS (("Goal" :IN-THEORY (ENABLE |ARRAY-SIZE-0<SIZE(S)>|)))
         :RULE-CLASSES NIL)

; USER ASSISTANCE MAY BE REQUIRED:

(DEFTHM
 ACL2-TOP-INV$INV
 (IMPLIES (INSERTION-SORT$INPUT-HYPS IN)
          (G :ASN (ACL2-TOP-INV IN)))
 :HINTS (("Goal" :IN-THEORY (DISABLE |_N_10$LOOP|)
                 :USE (ACL2-LOOP-INV-FLATSEQUENCE_N_13$INV LEMMA-2-ACL2-LOOP))))
```

D Insertion Sort Theorem File

```
(IN-PACKAGE "ACL2")

(INCLUDE-BOOK "insertion-sort-fns")

(LOCAL (INCLUDE-BOOK "insertion-sort-work"))

(SET-ENFORCE-REDUNDANCY T)

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

E Exported Properties of insert.vi

```
(encapsulate (((insert * *) => *))
  (local (defun insert (item arr)
           (cond ((endp arr)
                  (list item))
                 ((<= (car arr) item)
                  (cons (car arr)
                        (insert item (cdr arr))))
                 (t (cons item arr)))))

  (local (defthm insert-into-empty-list-is-sorted
     (sortedp (insert foo nil))))

   (local (defthm inserted-item-is-member
            (member foo (insert foo bar))))

   (local (defthm insert-preserves-membership
            (implies (member foo bar)
                     (member foo (insert foobar bar)))))


  (local (defthm insert-is-1-bigger
           (equal (len (insert foo bar)) (1+ (len bar)))))

  (local (defthm insert-preserves-how-many
           (let ((inserted-list (insert foo bar))
                 (unioned-list (cons foo bar)))
             (equal (how-many (perm-counter-example unioned-list inserted-list)
                              unioned-list)
                    (how-many (perm-counter-example unioned-list inserted-list)
                              inserted-list)))))

  ;;;
  (defthm insert-is-sorted
    (implies (and (true-listp foo)
                  (sortedp foo))
             (sortedp (insert bar foo)))
    :hints (("Goal" :in-theory (enable true-listp-len-0-is-nil))))

  (defthm insert-is-permutation
    (perm (cons foo bar)
          (insert foo bar)))
  (defthm insert-preserves-true-listp
    (implies (true-listp bar)
             (true-listp (insert foo bar))))
)
```

```
public int fib(int i)
{
    if(i <= 1)
        return 1;
    else
        return fib(i - 1) + fib(i - 2)
}
```

Figure 10: Fibonacci program in Java

```
(defun fib (i)
       (if (or (zp i) (= i 1))
           1
           (+ (fib (- i 1)) (fib (- i 2)))))
```

Figure 11: Fibonacci program in ACL2