# An approach to translate LUSTRE code to ACL2

by

Ruchi Dhingra

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Simanta Mitra, Co-major Professor
Suraj Kothari, Co-major Professor
Fritz Keinert

Iowa State University

Ames, Iowa

2005

Graduate College
Iowa State University

This is to certify that the master's thesis of

Ruchi Dhingra

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

# TABLE OF CONTENTS

## Abstract

In this thesis, we propose an approach to translate LUSTRE code to ACL2 code. The languages are very different and the translation is non-trivial. We have also validated our approach by translating and comparing results for many non-trivial test cases.

There is more than one way to perform the translation. Although, the proposed approach has performance inefficiencies – these do not matter because the main reason for the translation is to prove properties of the model. The proving operation does not require execution of the programs. The proposed approach has some key benefits such as it can be verified by execution – and that the resulting ACL2 code structure matches the input LUSTRE code structure. This makes it easier to use the theorem prover to prove properties of the program.

This translation work is important in the real-world. For example, it will allow Rockwell Collins to verify the correctness of models developed in Simulink®, SCADE® and LUSTRE using ACL2. The aforementioned tools are widely used in the aviation industry – in particular by Airbus and Boeing.

# 1. Introduction

In this section, we provide an overview of the research problem and its significance. We then present a summary of the work that we have done for this thesis.

## 1.1 The Research Problem

Reactive systems form an important class of computer systems and are most often used for representing critical software for control systems in domains such as aviation, space and satellite systems, locomotives, nuclear energy systems etc. LUSTRE is one of the most commonly used languages in this domain. SCADE®, a tool based on LUSTRE, is the European standard for the aviation software domain. In general, due to the safety critical nature of these systems, verification of correctness of models developed in Simulink®, SCADE®, and LUSTRE is of widespread interest. LUSTRE based reactive systems have been verified/validated using testing or model checking.

There have also been efforts to use the theorem prover PVS to verify properties of LUSTRE models. Although, conversions to the PVS theorem prover have been available – there is interest in using ACL2 theorem prover because anecdotal evidence suggests that the proving mechanism is faster using ACL2. Our literature survey did not yield any information on translation efforts from Lustre to a functional language.

## 1.2 Work Done

This work is a part of an overall project to build an automated translator from LUSTRE to ACL2 and to compare theorem proving experience across PVS and ACL2 and can be considered to be a proof of concept that demonstrates the feasibility and correctness of the translation while preserving code-readability.

To better understand the issues involved in the translation, we first undertook the tasks of manually translating a toy LUSTRE problem into imperative pseudo code, creating example codes in LUSTRE and trying to understand the C code generated by the compiler, and manually translating LUSTRE to ACL2 using an approach similar to that used for translation to PVS.

The focus of the effort has been at developing translation rules for a set of LUSTRE statements. In this thesis, we show one way of converting LUSTRE code to ACL2 code. After evaluating alternative translation approaches (state-based vs. stream-based and declarative vs. execution styled. These are described later in the thesis.), we used a stream-based execution style approach since it best matched our requirements.

Once we had identified the key issues to be addressed in our translation, the actual translation rules were defined. To validate these rules, we studied the validation approaches (semantic and execution based) and decided to using testing. We followed a bottom-up testing approach and ensured that our test suite included cases of reasonable complexity.

In addition to this thesis work, two technical reports (currently in their draft versions) are being developed: 'An Introduction to Theorem Proving' and 'A Comparison of Theorem Provers'.

'An Introduction to Theorem Proving' is a primer on theorem proving terms and concepts. Today, there are numerous provers that have been developed and are being supported by research communities and there is extensive documentation available on these large bodies of work. However, there is little organization or standardization of information in this domain. This, coupled with the fact that the basic concepts of the area are hard to grasp, leads to a very high entry barrier and learning curve for any newcomer. This technical report is intended to lower the entry barrier for students and researchers who are interested in this area of theorem proving.

The other technical report being developed, 'A Comparison of Theorem Provers' focuses on the theorem proving tools available today. It provides a quick, multi-dimensional comparison of a few theorem provers based on a few key parameters that we have identified to be meaningful and relevant as a basis for comparison. This comparison will serve as a handy introduction to some of the best theorem provers around today and can be used as a guide to shortlist theorem provers appropriate for the proof application being contemplated.

The remainder of this thesis is organized in the following manner. In section 2, we review related works and provide an introduction to LUSTRE and ACL2. In section 3, we detail our approach to translation and highlight the key issues in this regard. Section 4 details the translation rules. Section 5 discusses the validation effort. We conclude this report with a summary of the key results and our future plans.

# 2. Background

We proceed by presenting brief descriptions of LUSTRE and ACL2, followed by a literature survey of existing work.

## 2.1 Brief Introduction to LUSTRE

In this section we provide an introduction to the basics of LUSTRE. We focus on semantic aspects since our main interest is in the translation to ACL2. More comprehensive information can be found in [1, 2, 6].

LUSTRE is a synchronous dataflow language designed for programming reactive systems such as automatic control and monitoring systems, as well as for describing hardware. The dataflow aspect of LUSTRE makes it very close to usual description tools (block-diagrams, networks of operators, etc.) in these domains and its synchronous interpretation makes it well suited for handling time in programs. Moreover, this synchronous interpretation allows it to be compiled into an efficient sequential program.

LUSTRE is a functional language operating on streams (a finite or infinite sequence of values). All the values of a stream are of the same type, which is the type of the stream. A LUSTRE program or subprogram is called a node. A node defines one or several output stream parameters as functions of one or several input stream parameters. More details on this can be found in section 2.1.2.

A program has a cyclic behavior. The same code executes (repeatedly) at every instant of the clock. At the nth execution cycle of the program, all the involved streams take their current value to compute results which will be used for the n+1th execution cycle. Variables may reference each other's values. To allow benign cyclic dependencies, a delay operator (pre) is used. This operator returns the value of an expression, delayed by one instant. This helps avoid race conditions.

Clocks describe the cyclic behavior of a program [10]. Every program has a basic clock that corresponds to every instant in which inputs are received. From this clock, it is possible to define a slower clock by using a boolean stream. The clock defined by this stream is all of the instants when the stream takes the value '*true*'. Thus, in the table below, at every instant when the boolean stream A is '*true*', a slower clock (A Time Scale) is defined.

| Basic Time Scale | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | T | F | F | T | T | F | T | F | T | F |
| A Time Scale | 1 | | | 2 | 3 | | 4 | | 5 | |
| B | F | | | T | F | | T | | F | |
| B Time Scale | | | | 1 | | | 2 | | | |

It is also possible to define a slower clock that depends on the A time scale. For example, suppose that we have another clock B which is dependent on A. Then the time scales of the streams would be as shown in the above table. In general, it is possible to define an arbitrary hierarchy of clocks dependent on each other.

Note that clocks are not bound to physical time; LUSTRE considers time only in instants and has no predefined notion of real time.

### 2.1.1 Basic Operations

A LUSTRE program describes stream functions whose values can be recursively defined. Those functions are given by sets of equations defining output and local streams using expressions built over input, output, and local streams; constant streams; stream primitive operators; and user defined functions.

Constants denote constant streams; for instance '*true*' denotes the stream [t,t,t,...]. Variables denote streams; for instance '*x*' denotes [x0, x1, x2, ...]. Usual operators operate point wise over streams.

A unit delay operator followed-by '*fby*' (actually ->pre in LUSTRE) can be represented by the diagram:

| x | $x_0$ | $x_1$ | $x_2$ | ... |
|---|---|---|---|---|
| y | $y_0$ | $y_1$ | $y_2$ | ... |
| x fby y | $x_0$ | $y_0$ | $y_1$ | ... |

These constructs represent the core of LUSTRE. Complex behaviors can be easily created using this simple language, for instance:

**An integrator**
y = (0 fby y) + x

whose behavior is depicted in the diagram:

| x | $x_0$ | $x_1$ | $x_2$ | ... |
|---|---|---|---|---|
| y | $x_0$ | $x_0 + x_1$ | $x_0 + x_1 + x_2$ | ... |
| 0 fby y | 0 | $x_0$ | $x_0 + x_1$ | ... |

**A clock divider**
c = true fby (not c)

whose behavior is depicted in the diagram:

| not c | *f* | *t* | *f* | ... |
|---|---|---|---|---|
| true fby (not c) | *t* | *f* | *t* | ... |

These examples illustrate the use of the delay operator in building recursive stream definitions without deadlocks. The LUSTRE compiler performs static checks ensuring deadlock freedom.

### 2.1.2 Advanced Operations

**Nodes**
Convenient systems of equations can be saved into user-defined functions called nodes, which can be elsewhere used in expressions. For instance:

```
node integer (x : int) returns (y : int)
let
     y = (0 fby y) + x ;
tel
...
...
z = integer (x fby z) + u
```

Node parameters are typed and the compiler performs the usual type checks. Furthermore, the compiler checks for functionality: any local and output stream of a node should have one and only one defining equation. As mentioned earlier, the compiler also checks for proper clock use.

**Multi sampling**
It may be that slow and fast processes coexist in a given application. A sampling (or filtering operator) '*when*' allows fast processes to communicate with slower ones:

| c | *f* | *t* | *f* | *t* | ... |
|---|---|---|---|---|---|
| x | $x_0$ | $x_1$ | $x_2$ | $x_3$ | ... |
| x when c | | $x_1$ | | $x_3$ | ... |

Conversely, a holding mechanism, '*current*' allows slow processes to communicate with faster ones:

| c | *f* | *t* | *f* | *t* | ... |
|---|---|---|---|---|---|
| x | $x_0$ | $x_1$ | $x_2$ | $x_3$ | ... |
| y | | $y_0$ | | $y_1$ | ... |
| current(c,x) y | $x_0$ | $y_0$ | $y_0$ | $y_1$ | ... |

As we can see in the diagrams above, '*when*' discards its input '*x*' when the input condition '*c*' is '*false*'. Conversely, '*current*' fills the holes created by '*when*' with the input value '*y*' it got the last time the condition '*c*' was '*true*', if any, and otherwise with an initializing

sequence '*x*'. A free use of these primitives can yield overflow memory effects. Here also, the compiler checks for bounded memory. Those checks are called 'clock calculus' and are essential to ensure proper working of LUSTRE programs.

**If-Then-Else Expressions**

'*If-Then-Else*' expressions in LUSTRE provide the notion of selection. There is no distinction between statements and expressions in LUSTRE (like there is in C), so you can place '*if-then-else*' expressions almost anywhere, as shown below:

```
if (if b then (x and y) else (z1 and z2)) then
         if (e) then
            c + if (foo) then 1 else 0;
         else f;
      else
         if (g) then d else d +1;
```

These concepts conclude our overview of LUSTRE. More comprehensive information can be found in [1, 2, 6].

## 2.2 Brief Introduction to ACL2

ACL2 is both a programming language in which you can model computer systems and a tool to help you prove properties of those models. ACL2 is a very large, multipurpose system. You can use it as a programming language, a specification language, a modeling language, a formal mathematical logic, or a semi-automatic theorem prover. ACL2 is a mathematical logic together with a mechanical theorem prover to help you reason in the logic. Here we briefly describe some of the major components of ACL2 that were needed for our translation. More comprehensive information can be found in [7, 8, 14].

The ACL2 programming language is an extension of a non-trivial subset of Common Lisp. The subset contains none of the Common Lisp features that involve side effects. Thus, ACL2 focuses on the functional or applicative or side-effect free subset of Common Lisp.

The theorem prover supports first order logic and is an 'industrial strength' version of the Boyer-Moore theorem prover, Nqthm [8]. Models of all kinds of computing systems can be built in ACL2, just as in Nqthm, even though the formal logic is Lisp. Once you've built an ACL2 model of a system, you can run it. You can also use ACL2 to prove theorems about the model.

### 2.2.1 Basic Data Types

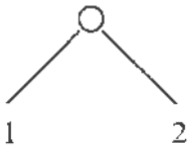
ACL2 supports five disjoint types of data objects:

- numbers
- characters
- strings
- symbols
- conses

The naturals (non-negative integers) are built from 0 by the successor function. The negative integers are built from the naturals by the negation function. The rationals are built from the integers by division. The complex numbers are built from a pair of rationals.

Similarly, a character can be constructed from a natural numbers (less than 256). A string can be constructed from a finite sequence of characters. A symbol can be constructed from two strings, one naming the package of the symbol and the other naming the symbol within that package.

Conses are the most commonly used objects in ACL2. They are sometimes called '*lists*', '*cons pairs*', '*dotted pairs*' or '*binary trees*'. Any two objects may be put together into a '*cons pair*'. The '*cons pair*' containing the integer 1 and the integer 2 is shown in the figure below. The left hand constituent is called the '*car*' of the pair. The right hand constituent is called the '*cdr*'.




### 2.2.2 Expressions

ACL2 programs are composed of expressions, also called '*terms*'. While many programming languages have expressions, statements, blocks, procedures, modules etc., ACL2 has just expressions.

A simple expression is:

- a variable symbol
- a constant symbol
- a constant expression, or
- the application of a functions expression, f, of n arguments, to n simple expressions, a1, ...an, written (f a1 ...an).

Comments may be written where white space is allowed. A comment begins with a semi-colon and ends with the end of the line. The following example expression contains the variable symbol '*date*', the constant symbol '*nil*', two constant expressions (a list and a string), and two function applications (of the function symbols '*if*' and '*equal*').

```
(if     (equal date '(august 14 1989))      ; Comments are written
        'Happy Birthday, ACL2!!!'           ; like this.
        nil)
```

### 2.2.3 Special Forms

ACL2 provides many built in constructs. Some of the commonly used ones include:

**let**

A '*let*' expression binds its local variables '$v_i$', in parallel to the values of the '$x_i$' and evaluates its body.

```
(let    ((v1 x1)
         ... (v2 x2)
          ....
         (vn xn))
        body)
```

**let ***

A '*let **' expression binds its locals sequentially and evaluates its body.

```
(let *  ((v1 x1)
         (v2 x2)
          ...
         (vn xn))
        body)
```

### 2.2.4 Primitive Functions

In this section we list many of the primitive functions of ACL2.

| Boolean Function | Description |
|---|---|
| (and p1 p2 ...) | Logical conjunction operator |
| (or p1 p2 ...) | Logical disjunction operator |
| (implies p q) | Logical implication |
| (not p) | Logical negation |
| (iff p q) | Logical equivalence |

| Arithmetic Function | Description |
|---|---|
| (integerp x) | Recognizer for integers |
| (rationalp x) | Recognizer for rationals |
| (complex-rationalp x) | Recognizer for complex numbers |
| (zerop x) | x = 0 |
| (zip x) | x = 0 or x is not an integer |
| (zp x) | x = 0 or x is not a natural |
| (< x y) | Less than relation |
| (<= x y) | Less than or equal relation |
| (> x y) | Greater than relation |
| (>= x y) | Greater than or equal relation |
| (+ x1 x2 ...) | Addition |
| (* x1 x2 ...) | Multiplication |
| (- x y) | Subtraction |
| (- x) | Arithmetic negation |
| (/ x y) | Division |
| (1- x) | Decrement by 1 |
| (1+ x) | Increment by 1 |
| (numerator r) | Numerator part of a rational |
| (denominator r) | Denominator part of a rational |
| (realpart c) | Real part of a complex |
| (imagpart c) | Imaginary part of a complex |
| (complex r i) | Make complex number with rational components |

| Cons Pairs and Lists | Description |
|---|---|
| (cons x y) | Construct an ordered pair |
| (car pair) | First component of a pair |
| (cdr pair) | Second component of a pair |
| (list x1 ... xn) | Linear list of n objects |
| (list* x1 ... xn z) | Linear list of n objects with z as the last cdr |
| (caar pair) | Car of the car |
| (cadr pair) | Car of the cdr |
| (cdar pair) | Cdr of the car |
| (cddr pair) | Cdr of the cdr |
| ... | ... |
| (cddddr pair) | Cdr of the cdddr |
| (append x1 x2 ...) | Concatenate linear lists |
| (nth n lst) | Nth element of a list (0-based) |
| (length lst) | Length of a list |
| (len lst) | Length of a list |

These concepts conclude our overview of ACL2. More comprehensive information can be found in [7, 8, 14].

## 2.3 Related Work

Reactive systems form an important class of computer systems and are most often used for critical software for control systems in domains such as aviation, space and satellite systems, locomotives, nuclear energy systems etc. LUSTRE is one of the most commonly used languages in this domain. For example, SCADE®, a tool based on LUSTRE, is the European standard for the aviation software domain. (Simulink® is the standard in America.) So far, LUSTRE based reactive systems have been verified/validated using testing or model checking. Recently, researchers have begun shown interest in theorem proving of LUSTRE code.

The SCADE® Suite from Esterel Technologies is a mature, commercially-supported toolset that is tailored to building reactive, safety critical-systems [10, 18, 19]. It has been used for creating commercial and military avionics systems. In SCADE®, software is specified using a combination of hierarchical state machines and block diagrams. These diagrams are automatically translated into LUSTRE. Because of the syntax of LUSTRE and the checks of the compiler, the composition of these diagrams always yields a complete, deterministic function.

Although SCADE® is a very capable toolset it lacks powerful analysis capabilities since it is not integrated with any theorem provers such as PVS or ACL2 [10]. Adding such capability would provide greater confidence in the correctness of specifications and code automatically generated by SCADE®. Rockwell Collins has shown interest in such a project. So far, they have undertaken the translation and verification of LUSTRE code using PVS [10].

The above translation approach considers a LUSTRE state to be a point of observation of certain quantities of interest in the system. The system variables represent the quantities of interest, i.e. they are mappings from states (the observation points) to values of those

quantities (at those observation points). When the system responds to changes in its environment, it moves to a new observation point i.e. a new state. So each state has an associated (finite) history of observations up to that point. The system specification is a set of constraints on the histories of the observations at each state. Properties of interest are similar to constraints but one has to establish that these are true. Verification entails proving implication of the property by the constraints in the specification.

GLOUPS [15] is an ongoing effort at building an automated translator from LUSTRE to PVS and at verification using PVS. They address the problem of verifying invariant properties (safety properties) of LUSTRE programs. Such properties can be expressed as LUSTRE observers (programs), and then reduced into a set of scalar proof obligations which are discharged into the PVS theorem prover. An (interactive) proof of these obligations is a proof of the initial invariant. Work on this project is still in its initial stages.

ACL2 is a relatively recent theorem prover that is appearing to gain popularity in industry. There is anecdotal evidence of its ease of use, efficiency and performance speed when compared with PVS. Unlike PVS, it supports execution and declarative styles of analysis. It has been successfully used in other domains [7, 8, 14] but there is no documentation of its use in verification of code written in dataflow languages.

Earlier research efforts in the area of verification of LUSTRE code have focused on validation using simulation and automated testing and on verification using model checkers [5]. Popular testing tools for validation of LUSTRE code include LUTESS [3] and LURETTE [12].

Model checking tools such as LESAR [11, 16], NBac [17], NP-Tools [9] are all based on exhaustive enumeration and past industrial example efforts have focused on design verification. LESAR is a verification tool dedicated to LUSTRE programs. It traverses the set of control states of a validation program either ennumeratively or symbolically. It restricts its search to the part of the program that can influence the satisfaction of the property. This is

an important feature since experience shows that in many practical cases the addressed property only concerns a very small part of the program: in such a case LESAR may be able to verify a property even if the entire state space of the program cannot be built [5].

NP-TOOLS is a general purpose formal verification toolbox for combinatorial circuits. Lucifer is the translator from LUSTRE to NP-TOOLS. The technical difficulty in the translation is that LUSTRE models dynamic systems i.e. systems whose outputs depend on the current inputs and all previous inputs while NP-TOOLS deals with static systems whose outputs depend only on current inputs. The dynamics of a LUSTRE model have to be modeled in NP-TOOLS by representing each LUSTRE node as an NP-TOOLS macro with additional inputs corresponding to the memory of the node [9].

Verification of LUSTRE code using theorem provers was largely limited to design / specification verification – not code verification. [9] provides a summary of many such works. To the best of our knowledge the only two significant efforts at verification of LUSTRE code have been using PVS. One of these was undertaken at Rockwell Collins and the other is the GLOUPS project undertaken at Verimag.

Our literature survey did not yield any information on translation efforts from LUSTRE to any functional language, including ACL2.

# 3. Approach

In this chapter, we discuss some of the alternative approaches to the translation.

There were two related translation efforts from LUSTRE available to us for study. One translation effort was from LUSTRE to C code. LUSTRE is a synchronous dataflow language and is declarative in nature (listed order of equations does not matter) whereas C is an imperative language and procedural in nature. The purpose of this existing translation was to execute the LUSTRE code. This C code was available from the LUSTRE compiler – but the details of the translation process were not available in the literature.

Another translation effort was from LUSTRE to PVS [10]. PVS is a specification language and a theorem prover which supports higher order logic. The purpose of this translation was to prove properties of the LUSTRE code using the PVS theorem prover. The translation was performed in a declarative style that is well supported by PVS. Reasoning about code is carried out by first modeling the system behavior as axioms and specifying the property of interest (such as a safety property) as a Lemma. Then, the theorem prover is used to help prove whether the safety property holds for the model.

Both of the above translation efforts are different from this work for the following reasons:

1. Unlike the translation to C, the purpose of this translation is to prove properties of the original LUSTRE program. Thus, there is a strong need for the translated code to be understandable and to have a similar structure as the original LUSTRE code.
2. Unlike C, ACL2 is purely functional language. ACL2 is a side-effect free version of Lisp and is quite restrictive. As detailed later, every code is checked automatically by ACL2 and then either rejected or accepted based on whether the code fulfills some properties. Even syntactically correct ACL2 code can be rejected. This makes it harder to program.
3. Unlike PVS, ACL2 specification logic is a first order logic. As discussed later, this makes it harder to translate LUSTRE code in a declarative manner to ACL2 code.

4. Unlike PVS, ACL2 code is also executable. This opens up the possibility of validating the effort by actually running the ACL2 code generated and comparing with results from the original LUSTRE code.

We realized that our choices are essentially along two orthogonal dimensions:

1. Modeling Style: state-based or stream-based.
2. Representation style: declarative or executable. (Usually executable could be further delineated into imperative, functional, and OO. However, ACL2 is functional).

In the remainder of this chapter, we first provide an overview of the different modeling and representation styles (in section 3.1) and then in section 3.2 we discuss our approach.

## 3.1 Modeling and Representation Styles

### 3.1.1 State-based Modeling

In state-based modeling, the system model comprises of states and transitions. A state is a sequence of executable program statements. A transition is a mapping from a state $P_i$ to another state $P_j$ (or to itself). These transitions are triggered when the value of boolean state variables are changed as described by the transition function ($\sigma$) . Note that the LUSTRE compiler uses this approach.

For example (see [4] for details), consider the following LUSTRE program where '*b*' is an input variable:

```
node example_state (b : int) returns (c: bool)
let
      c = false -> b and not pre(c);
tel
...
z = example_state (y)
```

We can consider the program to be equivalent to a model that has three states P0, P1, and P2 (see the figure at the end of this subsection).

The initial state $P_0$ comprises of the following executable statements:

```
c = false -> b and not pc;
pc = nil fby c;
```

At the initial instant '*c*' takes the value '*false*' and '*pc*' takes the value '*nil*'. Thus, the initial transition function σ (independent of the input '*b*') is:

$P_0 \xrightarrow{\sigma} P_1$ where `σ(c) = false, σ(pc) = nil`

The state $P_1$ has the following statements:

```
c = b and not pc;
pc = false fby c;
```

Now,

- if the input '*b*' is '*false*' then

  $P_1 \xrightarrow{\sigma} P_1$ where `σ(c) = false, σ(pc) = false`

- if the input '*b*' is '*true*' then

  $P_1 \xrightarrow{\sigma} P_2$ where `σ(c) = true, σ(pc) = false`

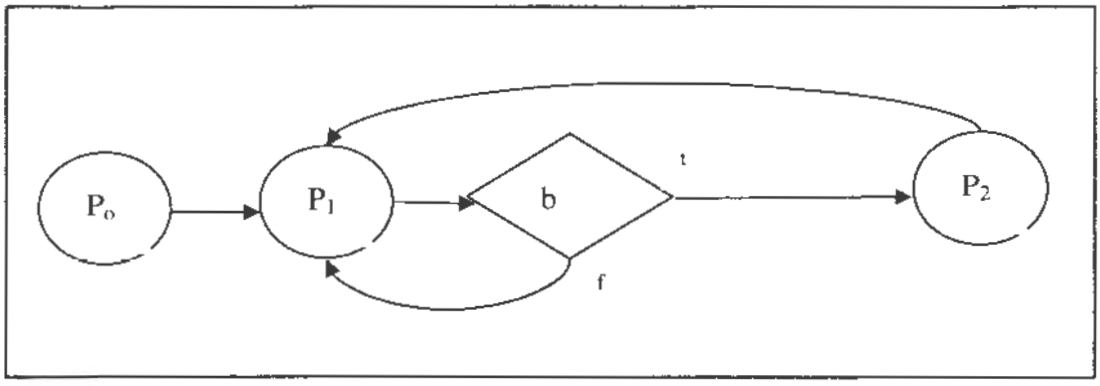
The state $P_2$ has the following statements:

```
c = b and not pc;
pc = true fby c;
```

Finally, we have the case that whatever the input b,

$P_2 \xrightarrow{\sigma} P_1$ where `σ(c) = false, σ(pc) = true.`

This gives us the finite automata as shown below:




To summarize, the entire program is represented by a state diagram with states representing sequence of statements and transitions representing changes in values of Boolean variables.

### 3.1.2 Stream-based Modeling

In this approach [10], a state is an observation of all variables that represent a system. At each clock instant, a new state is generated with the updated values of each variable. This leads to an infinite sequence of states. The history of each variable as it transitions through these states is modeled as a stream. LUSTRE and the PVS translation [10] both follow such a model.

The following LUSTRE code demonstrates this approach:

```
node example_stream (x, y : int) returns (z: int)
let
      z = x fby y;
tel
```

The values taken by input variables '*x*' and '*y*' are modeled as streams:

```
x = 1, 1, 1, 1…
y = 2, 4, 6, 8…
```

When the clock is at time = 0, the system is in its initial state and the variable values are:

```
x = 1
y = 2
z = 1
```

At the next state (when clock = 1) , the system transitions to the next state where the variables values are:

```
x = 1
y = 4
z = 2
```

The output variable '*z*' is also modeled as a stream:

```
z = 1, 2, 4, 6...
```

### 3.1.3 Declarative Programming

In a declarative style, properties about the program are expressed as assertions or predicates. These assertions describe '*what*' the system will do; not '*how*' it will do it. Thus the behavior of the program is expressed in terms of properties and not executable lines of code.
The following PVS code demonstrates this approach:

```
dec_example : THEORY [ a, b :sequence[real] ]
BEGIN
      c : sequence [real]
      c_axiom : AXIOM c = a + b
END dec_example
```

Note that c = a + b is modeled as an axiom i.e. 'c is the sum of a and b' is taken to be an axiom (a property that is to be considered true). This specification is not executable, no addition is actually being performed, and the value of c is not computed.

### 3.1.4 Executable Programming

In an execution style, the behavior of the program is modeled as a sequence of computations. The focus is on 'how' to perform a computation (or a set of computations) given an input; not on 'what' are the semantics of these computations. The execution style could be imperative, functional, and object-oriented. Here a distinction is attempted between the PVS declarative approach and the ACL2 executable approaches. PVS code cannot be executed in general.

## 3.2 Our Approach

The purpose of this translation work is to allow developers to use the ACL2 theorem prover to prove properties of the original LUSTRE program. Hence, it is very important to maintain a close correspondence between the structures of the original and translated codes. This includes using same variable names and method names. State based modeling totally changes the representation and does not meet this key criterion.

Another objective of the work was to allow a easy way to validate the translated code. ACL2 enables this because the code is executable and it is possible to directly compare results of running the translated code and the original LUSTRE code. Note that it is possible to translate to a 'declarative' form in ACL2 – by converting statements to be properties of variables (as done in PVS). However, that would render the validation effort by execution to be impossible. Another consideration was that PVS is higher order logic, whereas ACL2 is a first order logic. This makes it harder to translate LUSTRE to ACL2 than to PVS using the declarative approach. Hence, we used a stream-based model and the executable representation style for our translation.

In a PVS like declarative approach, the prover generates many TCCs (type checking conditions) and proof obligations. These need to be proved in addition to the main properties of interest. The number of such TCCs generated is noticeable even in toy problems and can cause substantial increase in the proving effort and the time needed for industrial problems. This issue does not arise in the execution style approach where TCCs and proof obligations are not generated for trivial cases [1].

In a declarative approach, composing predicates most often require creation of intermediate variables unknown to the programmer. This makes the proving effort harder because the correspondence between the original code and the translated code is not maintained. In our approach, there is a close correspondence between the original LUSTRE code and the translated ACL2 code. Hence it is easier for the programmer to understand the ACL2 code and hence to confidently participate in the proving process.

In an execution style translation validation using testing is possible. We have demonstrated this in the next section. This provides confidence in the correctness of the code / translation and may be a better choice than validation for smaller, easier to prove properties. For more complex properties, or in situations where verification is needed / wanted, once an execution style code is generated, it can always be analyzed using invariants similar to the declarative approach. The converse is not possible.

Thus, our approach avoids some of the characteristic disadvantages of a declarative style while allowing us to utilize its benefits in future (when we so desire).

# 4. Translation from LUSTRE to ACL2

In this chapter, we first describe the rules for the translation process for converting LUSTRE code to ACL2 code and explain the process using a simple example. We conclude the chapter by providing a comparison of our translation effort with a similar translation effort from LUSTRE to PVS by Rockwell Collins in collaboration with the University of Minnesota [10].

As discussed previously, we have chosen to translate LUSTRE code to an executable model that mimics the LUSTRE format – rather than use a declarative model. One of the reasons for this choice is that we can directly compare results of running LUSTRE code and running the ACL2 translation code. Although ACL2 is lisp based, it is a theorem prover and there are several restrictions on coding which makes the translation/execution cumbersome.

Some of these restrictions are:

a) Global variables are not allowed i.e. the body may contain no free variables other than the formal arguments. In other words, programming has to be done in a purely functional style. This helps ACL2 prove properties – because of the absence of side-effects. However, this does mean inefficiencies because values must be re-computed as they cannot be saved anywhere. On the other hand, this inefficiency is irrelevant because the motive for translation is to prove properties of the code and not to execute it.

b) Any function used in the body of a function – other than the ones being defined – must have been introduced earlier. Thus, a system of definitions must be presented 'bottom up'. (Note that the ACL2 mutual-recursion method allows several functions to be defined simultaneously).

c) Recursive definitions must be proved to terminate.

Note that the system rejects any definition (although it may be syntactically correct) that does not follow the rules. In such a situation, the cause for the rejection must be identified by

going over the pages of proof attempts dumped by the system. After finding the cause, the code must be fixed – usually by finding an alternative way to represent it.

## 4.1 Translation Rules

### 4.1.1 Stream

Each stream is translated to a list of the same type. For example, a stream of integers becomes a list of integers and a stream of booleans becomes a list of booleans. The element 1 of the translated list represents value of the stream for the current clock tick, element 2 represents the value at the previous tick, element 3 represents the value at the tick previous to that and so on. The last element of the list represents the value at clock tick 1.

In LUSTRE the stream represents a set of infinite values. However, in practice, only a finite number of clock steps are executed. Although, we use a finite list to represent the steps, the ACL2 prover will prove properties for any arbitrary length of each list (one does not have to execute the programs to verify the properties of the program).

In LUSTRE every stream is associated with a clock. In the translated version, element0 represents the clock stream which was used to create this stream (in other words, the parent stream). The rest of the elements represent the data values of the stream. The default LUSTRE clock is represented by a '*nil*' – and clocks other than the default clock are ordinary streams and represented in the normal manner (by lists whose first element represents their clock). Note that our representation allows us to handle arbitrary depth of nesting of '*when*' and '*current*' operations.

Clocks are only required by the '*when*' and '*current*' constructs (and '*condact*'). A '*when*' construct (samples) introduces a clock to the stream and shrinks the stream to contain only elements corresponding to the elements of the clock stream which have the value '*true*'. A '*current*' construct (interpolates) reverts a stream back to the original clock. The sampled

stream can be sampled again – and thus there is a hierarchical system of clocks and this is well supported by our translation scheme.

One assumption is that the translation is done on compilable LUSTRE code. The compiler makes sure that the syntax is correct and that clock usage rules are followed. Another assumption is that the output of a node is not fed back to the input of the same node. This assumption is enforced by tools like SCADE®. This allows us to handle translation at the node level.

### 4.1.2 Data operators and 'if then else'

Operators such as +, - , and constructs such as '*if then else*' etc. have matching constructs in ACL2. In general, these constructs can be assumed to work on individual values within a node. Here the translation is straightforward.

There are several syntactic situations where the result of the operation must be directly fed into a function. Since functions operate on streams and not on individual values (because a function can use a previous value of a stream) – it may be necessary to create a list version of the operator. For example, the test case whenCurrent2 (see appendix) needed a ACL2Plus function to be defined.

### 4.1.3 Node

Each LUSTRE node is translated to a pair of ACL2 functions. The translation has a specific format as explained with a specific example in the following subsection. LUSTRE nodes can return multiple values and this is handled in ACL2 by returning a list of lists.

### 4.1.4 Utilities

A library of utilities has been created for use in the translated code. These are described in the following table.

| Utility | Description |
|---|---|
| (ACL2_get_clk l) | Returns the clock stream (which is element 0) from the stream l. |
| (ACL2_get_dat l) | Returns the data stream (which is list of elements 1, 2, … etc) . |
| (ACL2_mak_lst c d) | Creates a list given a clock stream and a data stream |
| (ACL2_cur_val l) | Returns data item for current clock tick |
| (ACL2_pre_val l i) | Returns data item from previous clock tick or i if undefined |
| (ACL2_add_val l c v) | Adds the new clock tick value v |
| (ACL2_rem_lst l i) | Returns the remainder of the list l |
| (ACL2_when l c) | Returns a new list which has c as the clock stream and only has data items for items in l corresponding to TRUE in clock c. |
| (ACL2_current l) | Returns a new list with values also defined for instances where l's clock has FALSE. (This is translation for LUSTRE's current). |

## 4.2 An Example of the Translation Process

In this section, we describe the translation process using a specific example. First, we present a LUSTRE code and given an input set – explain what we expect as output. Next, we present the translated ACL2 code. Key parts of this translated code are tagged. Finally, we explain the tagged parts.

### 4.2.1 LUSTRE Code

The example Lustre code shown here has two nodes: testWhen3 and fun_a. Given input streams '*y*' = 1 2 3 4 5, '*z*' = 9 10 11 12 13, and '*b*' = 1 0 1 0 1, we expect '*r1*' to be 15 15 10 10 14 and '*r2*' to be 15 15 12 12 16. This example has several key complexities: function

calls, multiple return values, use of clocks in the argument, use of '*when*' and '*current*', use of '*pre*' and operators.

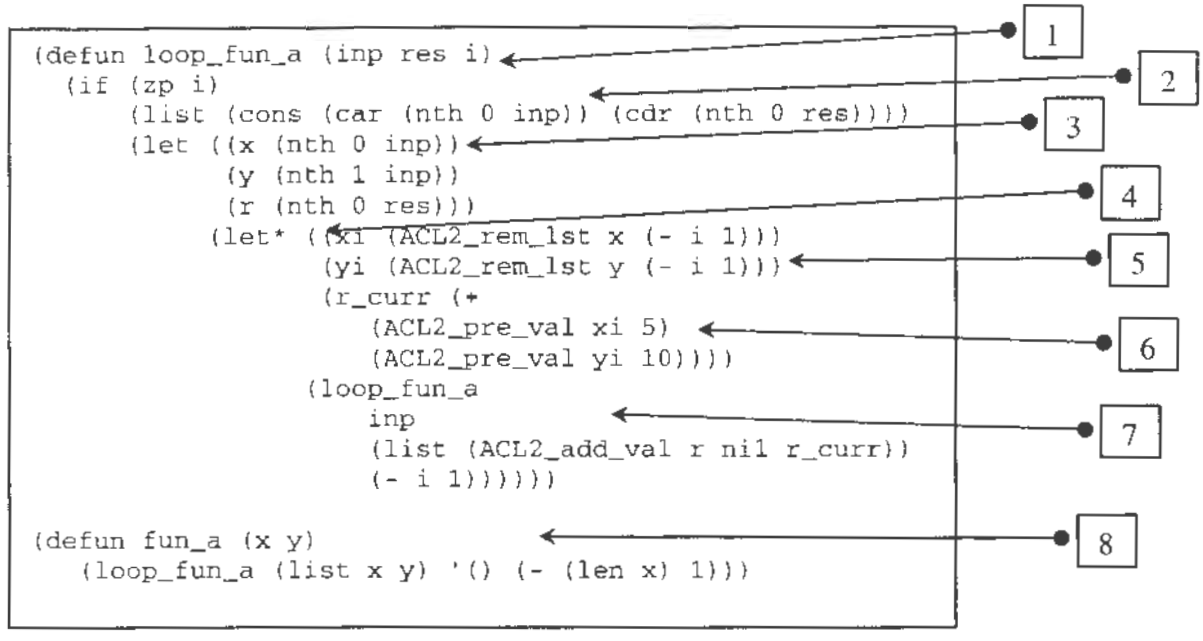
```
Node testWhen3(y, z: int; b: bool) returns (r1, r2: int);
let
      r1 = current (fun_a((y, z) when b));
      r2 = current (fun_a(y, z)  when b);

tel

node fun_a (x, y: int) returns (r: int);
let
      r =  (5 -> pre (x)) + (10 -> pre(y));
tel
```

### 4.2.2 Translated ACL2 Code

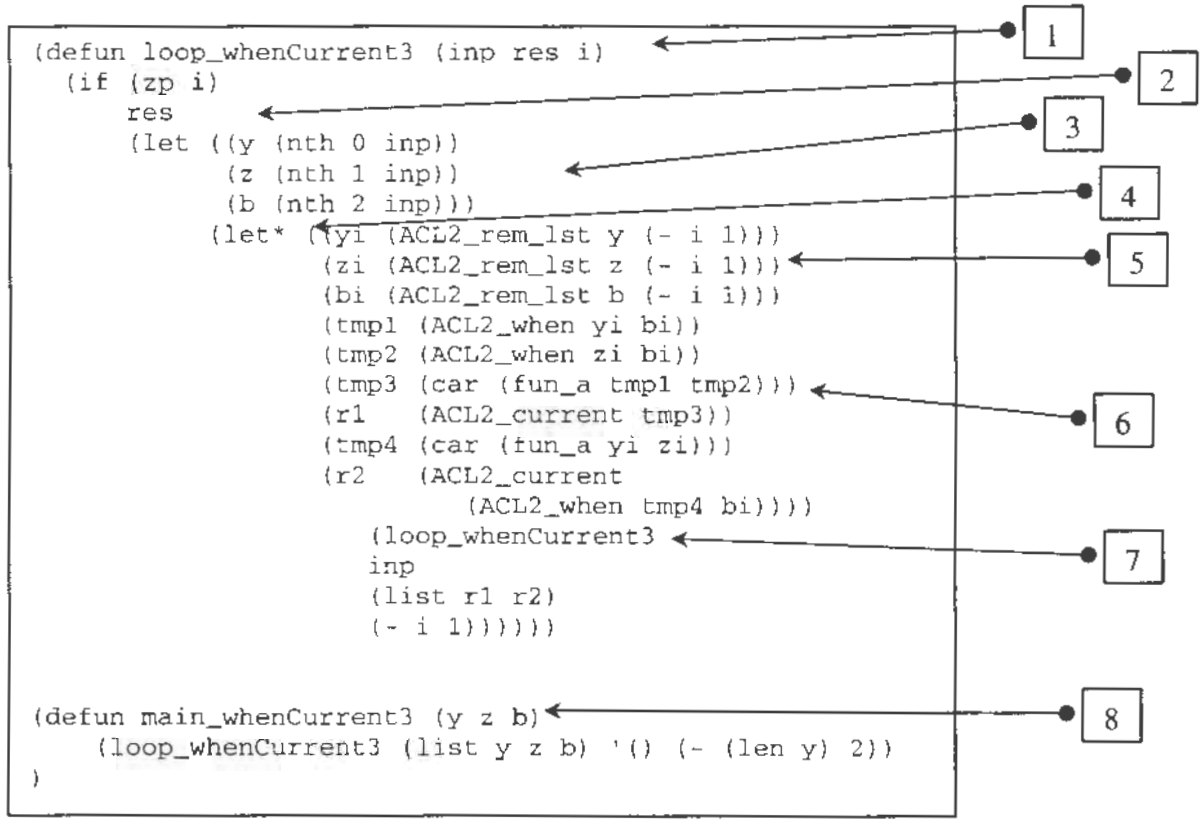
Here we have applied our translation rules on the above Lustre code to generate corresponding ACL2 code. For each LUSTRE node, we have a pair of ACL2 functions. Thus, we have a pair of functions (fun_a and loop_fun_a) corresponding to the node fun_a and a pair of functions (main_whenCurrent3 and loop_main_whenCurrent3) corresponding to the node testWhen3. The code here is tagged and the tags are explained later. To validate the translation, the translated code was executed in ACL2 environment and the results matched the results obtained from directly executing the LUSTRE code.



```
(defun loop_fun_a (inp res i)
  (if (zp i)
      (list (cons (car (nth 0 inp)) (cdr (nth 0 res))))
      (let ((x (nth 0 inp))
            (y (nth 1 inp))
            (r (nth 0 res)))
           (let* ((xi (ACL2_rem_lst x (- i 1)))
                  (yi (ACL2_rem_lst y (- i 1)))
                  (r_curr (+
                     (ACL2_pre_val xi 5)
                     (ACL2_pre_val yi 10))))
                 (loop_fun_a
                     inp
                     (list (ACL2_add_val r nil r_curr))
                     (- i 1))))))

(defun fun_a (x y)
   (loop_fun_a (list x y) '() (- (len x) 1)))
```





```
(defun loop_whenCurrent3 (inp res i)
  (if (zp i)
      res
      (let ((y (nth 0 inp))
            (z (nth 1 inp))
            (b (nth 2 inp)))
           (let* ((yi (ACL2_rem_lst y (- i 1)))
                  (zi (ACL2_rem_lst z (- i 1)))
                  (bi (ACL2_rem_lst b (- i 1)))
                  (tmp1 (ACL2_when yi bi))
                  (tmp2 (ACL2_when zi bi))
                  (tmp3 (car (fun_a tmp1 tmp2)))
                  (r1   (ACL2_current tmp3))
                  (tmp4 (car (fun_a yi zi)))
                  (r2   (ACL2_current
                           (ACL2_when tmp4 bi))))
                 (loop_whenCurrent3
                     inp
                     (list r1 r2)
                     (- i 1))))))


(defun main_whenCurrent3 (y z b)
    (loop_whenCurrent3 (list y z b) '() (- (len y) 2))
)
```

### 4.2.3 Explanation of Translated Code

Here we provide explanation of the translation process. The steps here correspond to the tags in the two diagrams above.

**Step 1:** Corresponding to every node in LUSTRE – there are two functions in ACL2. One has the same signature as the LUSTRE node. The other function is essentially a loop – which recurses and calculates the resulting output stream(s). The loop function has two, three, or four inputs: '*inp*', '*tmp*', '*res*', '*i*'. Note that in ACL2 one cannot store values in global variables. So, data is passed via the arguments list. '*inp*' stands for input arguments. This is a list of input streams. '*tmp*' stands for temporary values and is a list of temporary variables. '*res*' stands for list of result streams. '*i*' is a countdown variable and the recursion stops when the value of i reaches zero.

**Step 2:** The '*if*' part at the beginning of the loop function is used to terminate the recursion. If '*i*' is zero, then the results list is returned. Note that when the results have to be used for calculations, we have to more careful – and make sure that the correct clock is attached to the result. For the whenCurrent3 function – the results would just be printed – and so we could get away with just returning the results list. However for the fun_a we have to attach the parent clock from the input stream to the clock of the result streams.

**Step 3:** The '*let*' section is used to extract the individual streams from the input lists of streams. We can use a let because the order of execution is not important here.

**Step 4:** The '*let**' section is used to perform the actual computation part of the node. Here the order of execution is important. For the translation, it is important to first topologically sort the LUSTRE equations – so that the calculations that need to be performed first are done first.

**Step 5:** Usually, we need to extract the right length of the input streams for the particular iteration of the loop method. This is achieved using the ACL2_rem_lst utility. We do not

need to use this utility for '*tmp*' and '*res*' streams as these are in the process of being generated (whereas the input streams are given).

**Step 6:** This is where intermediate computations are carried out. Note how the previous value of a stream is retrieved. Also, note how the fun_a method is being called in the loop_whenCurrent3 example.

**Step 7:** Here the recursion occurs. The loop method calls itself and creates new arguments based on the results it generates. Note that '*i*' is decremented ensuring that the loop terminates.

**Step 8:** Corresponding to every node in LUSTRE – there are two functions in ACL2. This ACL2 function represents the LUSTRE node with the same signature. It calls the loop function to calculate the result stream(s).

## 4.3 Comparison with LUSTRE to PVS Translation

The following table lists the LUSTRE features that we have implemented and those that will be implemented in the next stage. It also provides a quick comparison to list of features included the LUSTRE to PVS translation undertaken by Rockwell Collins [10].

| LUSTRE Feature | Implemented in Our Translation | Implemented in PVS Translation |
|---|---|---|
| Assignment statement | Yes | Yes |
| Clocks | Yes | Yes |
| Current | Yes | No |
| fby | Yes | Yes |
| If-then-else | Yes | Yes |
| Nodes | Yes | Yes |
| Pre | Yes | Yes |

| | | |
|---|---|---|
| Stream | Yes | Yes |
| Tuples | Yes | Yes |
| When | Yes | No |
| Arrays | Future Work | No |
| Case expressions | Future Work | Yes |
| Condact | Future Work | Yes |
| User-defined data types, enumerated data types | Future Work | No |

The '*condact*' feature is easy to implement since it has a more limited scope than '*when*' and '*current*' which have already been implemented. Case statements and arrays in ACL2 are similar to those in LUSTRE and the translations are expected to be straight forward. These will be included in the next version. Although '*tuples*' are supported, we have not explored how to support enumerated and user defined data types.

# 5. Validation

We begin this chapter by providing an overview of validation approaches that we considered and then include our test plan and report.

## 5.1 Validation Approaches

### 5.1.1 Importance of Validating Translation Rules

Validation is a way to infuse confidence in the correctness of the translation. There are two primary reasons [13] for not blindly trusting translation from one high level language A (in our case LUSTRE) to another high level language (in our case ACL2):

- High level languages tend to be poorly defined and complex. The semantics are not user friendly and that makes it hard to know what a program means or what is supposed to happen when it is executed. This makes it difficult to know how a program written in a particular language should be translated to another language. In some cases, the only way to find out is to execute it and analyze the output.
- The translation process / software tends to be large and complex and may have bugs. Hence, even when we try and understand a particular code in one high level language, we are unsure that the translation will incorporate all the semantics of the source code.

Next, we explain the various validation approaches that were considered.

### 5.1.2 Validation Approach Alternatives

In order to validate a translation, one can compare the two sets of code, the LUSTRE source and the translated ACL2 code, by comparing their semantics or by comparing their outputs by executing them. These options are similar to what is widely accepted as validation alternatives for complier codes [13]. We summarize these options in the following subsections.

#### 5.1.2.1 Comparison of Outputs using Testing

Testing is an often used methodology for validating smaller systems / translations that are not too complex to analyze and reason about effectively without a very heavy investment in terms of time or resources.

We followed this approach to validate semantic equivalency of the two codes. Since exhaustive testing is not a feasible option, we designed our tests as a series of test cases that test the simplest structures and then more and more complex program structures. The idea is based on the principle of induction and this approach validates correctness of complex programs built using validated simpler structures. The test plan and report are outlined in section 5.2.

#### 5.1.2.2 Comparison Using Semantics

If one chooses to compare the semantics of each statement of the original code and its translation, the comparison can be based on axiomatic semantics, operational semantics, or denotational semantics [13]. *Considering the lack of time and the skills needed to use these approaches, we decided not to pursue them for our validation effort.* We list them here for completeness.

**Axiomatic Semantics**

This type of semantics defines a language by providing assertions and inference rules for reasoning about programs.

Assertions can be expressed as 'Hoare triples'. For example, the inference rule for an '*if*' construct might be given as something such as

$$\frac{\{ P \text{ and } e \}\ ct\ \{ Q \},\quad \{ P \text{ and not } e \}\ cf\ \{ Q \}}{\{ P \}\ \text{if } e \text{ then } ct \text{ else } cf\ \{ Q \}}$$

According to the rule, when reasoning about an '*if*' construct, to prove that the post-condition *Q* is established, it is sufficient to prove that it is established by '*ct*' whenever '*e*' holds, and by '*cf*' whenever '*e*' does not hold (in each of these cases, we assume that the common pre-condition '*P*' holds, too).

We can check for the functional equivalence of two programs by showing that starting from the same pre-condition they establish the same post condition.

**Operational Semantics**

This type of semantics defines a language in terms of the operation of a machine (possibly abstract) executing the program. This kind of semantics focuses on the implementation process. For example, it might define the meaning of an '*if*' construct such as **if** *e* **then** *ct* **else** *cf* in terms of labels and jumps

```
        < e >
        JUMP label1
        < ct >
        GOTO label2
label1:
        < cf >
label2:
```

where the program fragments in angle brackets should be replaced with their operational semantics definitions, recursively. '*JUMP*' transfers control to the relevant label if the previous expression evaluates to '*false*'; '*GOTO*' is an unconditional jump.

This kind of semantics does not do a good job of defining the actual meaning, as it is defined only in terms of another programming language, which itself needs a definition. Additionally, since LUSTRE is a synchronous dataflow language with nodes executing concurrently and ACL2 is a functional language, the translations to an appropriate common low-level language with same structure is harder to achieve.

**Denotational Semantics**

This type of semantics defines a language by mapping it to mathematics. This is accomplished by developing a mathematical model that defines 'meaning functions'. Each meaning function maps a language construct to a mathematical value. The value is thus the 'meaning' of the construct.

For example, the mathematical model suitable for an imperative language is one of states and state transitions. The denotation of an '*if*' construct in an imperative language would be defined in terms of the mathematical meanings of its component constructs:

$$D\ [|\ \textbf{if}\ e\ \textbf{then}\ ct\ \textbf{else}\ cf\ |]\ p\ s =$$
$$D\ [|\ ct\ |]\ p\ s, \quad \text{if } D\ [|\ e\ |]\ p\ s = \text{true}$$
$$D\ [|\ cf\ |]\ p\ s, \quad \text{if } D\ [|\ e\ |]\ p\ s = \text{false}$$

Here '*p*' is the environment, which is a mapping of program identifiers to abstract locations, and's' is the state, which is a mapping from abstract locations to mathematical values. '*D [| |]*' is a function that maps program language commands to a new state. Hence, it maps program syntax to mathematical values.

This approach of modeling abstract meanings of programs, independent of any language constructs and machine implementation. It satisfies the requirement that a program must have the same logical behavior in any language. It is also at the right level of abstraction needed to verify a translation process or software.

## 5.2 Test Plan and Report

A simplified version of LUSTRE can be described by the following grammar [4]:

```
program ::= equations
equations ::= equation | equation ; equations
equation ::= id = (clock) expression
expression ::= sexp | sexp -> sexp | k fby sexp | when (sexp, sexp) | current (sexp)
sexp ::= k | id | dataoperator (sexp, …., sexp)
clock ::= expression | true
```

As discussed earlier, we have used an inductive test plan based on structure. We began by testing the simplest structures and then more complex programs built up of the simple structures. Our reasoning is that any program is composed from these structures and based on the principle of induction these programs will be correct if the base structures are correct.

Our first series of tests use a single node and perform simple operations on data. The next set of tests has multiple equations and conditions. The third set of tests is more complicated and focuses on the use of tuples and multi-valued functions. The next set of tests exercise multi-node programs. The final set of tests focuses first on the complex sampling and delaying constructs ('*when*' and '*current*') and then on a few complex combinations of all the different statement types.

The test report is as outlined below. (Note that the actual test codes are included in the appendix).

| Test Set | Test Case No. | Test Case Title | Passed / Not Passed |
|---|---|---|---|
| 1 | 1 | Returning a constant output irrespective of the input | Passed |
| 1 | 2 | Returning the double of an integer stream input | Passed |
| 1 | 3 | Adding a constant to each value of an integer stream input | Passed |
| 2 | 4 | Execution of an if-then-else statement | Passed |
| 2 | 5 | Inserting a constant at the beginning of a stream | Passed |
| 2 | 6 | Multiple equations as part of a single node | Passed |
| 3 | 7 | Returning a constant output irrespective of the input with tuples | Passed |
| 3 | 8 | Execution of an if-then-else statement with tuples | Passed |
| 3 | 9 | Multiple equations as part of a single node with tuples | Passed |
| 4 | 10 | Multiple nodes | Passed |
| 5 | 11 | Basic test case for 'when' and 'current' | Passed |
| 5 | 12 | Second test case for 'when' and 'current' – 2 nodes | Passed |
| 5 | 13 | Third test case for 'when' and 'current' – 2 nodes and 'pre' | Passed |

# 6. Conclusions and Future Work

In this thesis, we have provided the translation rules for converting LUSTRE code into executable ACL2 code. The translation rules have been shown to work for a test case suite spanning simple statement structures and complex methods.

This work is supported by two technical reports – 'An Introduction to Theorem Proving' and 'A Comparison of Theorem Provers'. The former provides a primer to theorem proving terms and concepts while the latter is a multi-dimensional comparison of some of the most widely used theorem provers.

There are several areas of future work:

- Extension of the translation to include the remaining LUSTRE constructs.
- Building an automated translator from LUSTRE to ACL2 based on the rules provided in this thesis.
- Comparison of PVS and ACL2 theorem proving experiences.
- Validation of this translation using a more formal approach (such as using denotational semantics).
- Translation of LUSTRE to ACL2 using a declarative approach similar to that of the PVS translation.

# 7. References

[1] S. Bensalem, P. Capsi, C. Parent-Vigouroux and C. Dumas. A methodology for proving control systems with LUSTRE and PVS. Seventh Working Conference on Dependable Computing for Critical Applications (DCCA7) San Jose, USA, January 1999.

[2] A. Benveniste, P. Capsi, S. Edwards, N. Halbwachs, P Guernic and R. Simone. The Synchronous Languages 12 Years Later. Proceedings of the IEEE, Vol. 91, No. 1, 2003.

[3] L. Bousquet, F. Ouabdesselam, J.L. Richier and N. Zuanon. Lutess: testing environment for synchronous software. In Tool Support for System Specification Development and Verification. Advances in Computer Science, Springer, 1998.

[4] N. Halbwachs, P. Capsi, P. Raymond and P. Pilaud. The synchronous dataflow programming language LUSTRE. Proceedings of the IEEE, Vol. 7, No. 9, 1991.

[5] N. Halbwachs and P. Raymond. Validation of Synchronous Reactive Systems: from Formal Verification to Automatic Testing. ASIAN'99, Asian Computing Science Conference, Phuket (Thailand), December 1999.

[6] N. Halbwachs and P. Raymond. A tutorial of LUSTRE. www-verimag.imag.fr/~halbwach/PS/**tutorial.ps**. Last referenced February 27, 2005.

[7] M. Kaufmann, P. Manolios and J Strother Moore (eds.). Computer-Aided Reasoning: ACL2 Case Studies, Kluwer Academic Publishers, June, 2000. (ISBN 0-7923-7849-0)

[8] M. Kaufmann, P. Manolios and J Strother Moore. Computer-Aided Reasoning: An Approach, Kluwer Academic Publishers, June, 2000. (ISBN 0-7923-7744-3).

[9] O. Laurent, P. Michel and V. Wiels. Using formal verification techniques to reduce simulation and test effort. In Proceedings of FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001.

[10] S. Miller. Autocoding Tools Interim Report, Rockwell Collins Advanced Technology Centre, Cedar Rapids, USA, February, 2004.

[11] C. Ratel, N. Halbwachs and P. Rayomond. Programming and verifying critical systems by means of the synchronous dataflow programming language LUSTRE. In ACM-SIGSOFT'91 Conference on Software for Critical Systems, New Orleans, December 1991.

[12] P. Raymond, D. Weber, X. Nicollin and N. Halbwachs. Automatic testing of reactive systems. In 19th IEEE Real-Time Systems Symposium, Madrid, Spain, December 1998.

[13] Official webpage for ACL2. http://www.cs.utexas.edu/users/moore/acl2/. Last referenced February 27, 2005.

[14] Official webpage for GLOUPS. http://www-verimag.imag.fr/SYNCHRONE/index.php?page=gloups. Last referenced February 27, 2005.

[15] Official webpage for LESAR. http://www-verimag.imag.fr/~raymond/tools/lv4-distrib.html. Last referenced February 27, 2005.

[16] Official webpage for NBac. http://www.irisa.fr/prive/bjeannet/nbac/nbac.html. Last referenced February 27, 2005.

[17] Official webpage for SCADE® . http://www.esterel-technologies.com/products/scade-suite/overview.html. Last referenced March 25, 2005.

[18] SCADE® Language Reference Manual. Available on request from Esterel Technologies (www.esterel-technologies.com).

# Appendix - Test Cases LUSTRE and ACL2 Codes

**Test Set 1: Simple Test Cases:**

Test Case 1: Returning a constant output irrespective of the input

This test case takes an integer stream input y, ignores it and returns an integer stream x which is a constant 5.

The LUSTRE example was:

```
node constant (y: int) returns (x: int);
let
   x = 5;
tel
```

The ACL2 translation was:

```
(defun loop_constant (res i)
  (if (zp i)
      res
      (let ((r1 (nth 0 res)))
           (let* ((x 5))
                 (loop_constant
                    (list (ACL2_add_val r1 nil x))
                    (- i 1))))))


(defun main_constant (y)
   (loop_constant '() (- (len y) 1))
)
```

The test data was:

Input: 1 1 1 1

Output: 5 5 5 5

```
(cw '~x0~%'
        (ACL2_get_dat (car
           (main_constant (ACL2_mak_lst '() '(1 1 1 1))))))

(if (equal '(5 5 5 5)
           (ACL2_get_dat (car
              (main_constant (ACL2_mak_lst '() '(1 1 1 1))))))
  'passed constant test'
  'failed constant test')
```

Test Case 2: Returning the double of an integer stream input

This test case takes an integer stream input y and returns a stream in which each value is double of the corresponding value in y.

The LUSTRE example was:

```
node data (y: int) returns (x: int);
var z: int;
let
   z = 2 * y + y + y + y;
   x = (z - y)/2;
tel
```

The ACL2 translation was:

```
(defun loop_data (inp tmp res i)
  (if (zp i)
      res
      (let ((y (nth 0 inp))
            (z (nth 0 tmp))
            (res0 (nth 0 res)))
           (let* ((yi (ACL2_rem_lst y (- i 1)))
                  (y_curr (ACL2_cur_val yi))
                  (z_curr (+ y_curr (+ y_curr (+ y_curr (* 2 y_curr)))))
                  (x_curr (/ (- z_curr y_curr) 2)))
                 (loop_data inp
                            (list (ACL2_add_val z nil z_curr))
                            (list (ACL2_add_val res0 nil x_curr))
                            (- i 1)))))))


(defun main_data (y)
    (loop_data (list y) '() '() (- (len y) 1))
)
```

The test data was:

Input: 3 4 5 6
Output: 6 8 10 12

```
(cw '~x0~%'
    (ACL2_get_dat (car (main_data (ACL2_mak_lst '() '(6 5 4 3))))))

(if (equal '(12 10 8 6)
    (ACL2_get_dat (car (main_data (ACL2_mak_lst '() '(6 5 4 3)))))))
```

Test Case 3: Adding a constant to each value of an integer stream input

This test case takes an integer stream input y and returns a stream in which each value of y is incremented by a constant.

The LUSTRE example was:

```
node variable (x: int) returns (y: int);
var z: int;
let
   z = 5;
   y = x + z;
tel
```

The ACL2 translation was:

```
(defun loop_variable (inp tmp res i)
  (if (zp i)
      res
      (let ((x (nth 0 inp))
            (z (nth 0 tmp))
            (res0 (nth 0 res)))
           (let* ((xi (ACL2_rem_lst x (- i 1)))
                  (x_curr (ACL2_cur_val xi))
                  (z_curr 5)
                  (y_curr (+ z_curr x_curr)))
                 (loop_variable
                    inp
                    (list (ACL2_add_val z nil z_curr))
                    (list (ACL2_add_val res0 nil y_curr))
                    (- i 1)))))))

(defun main_variable (x)
   (loop_variable (list x) '() '() (- (len x) 1))
)
```

The test data was:

Input: 1 2 3 4
Output: 6 7 8 9

```
(cw '~x0~%'
    (ACL2_get_dat (car (main_variable (ACL2_mak_lst nil '(4 3 2 1))))))

(if (equal '(9 8 7 6)
    (ACL2_get_dat (car (main_variable (ACL2_mak_lst nil '(4 3 2 1))))))
  'passed variable test'
  'failed variable test')
```

**Test Set 2: Test Cases Involving Multiple Equations:**

Test Case 4: Execution of an if-then-else statement

This test case checks the implementation of an if-then-else statement.

The LUSTRE example was:

```
node conditional (x:int; y: int) returns (result: int);
let
   result = if (x > y)
                then x + y
                else y - x;
tel
```

The ACL2 translation was:

```
(defun loop_conditional (inp res i)
  (if (zp i)
      res
      (let ((x (nth 0 inp))
            (y (nth 1 inp))
            (res0 (nth 0 res)))
           (let* ((xi (ACL2_rem_lst x (- i 1)))
                  (yi (ACL2_rem_lst y (- i 1)))
                  (x_curr (ACL2_cur_val xi))
                  (y_curr (ACL2_cur_val yi))
                  (res_curr (if (> x_curr y_curr)
                                (+ x_curr y_curr)
                                (- y_curr x_curr))))
                (loop_conditional
                   inp
                   (list (ACL2_add_val res0 nil res_curr))
                   (- i 1)))))))
(defun main_conditional (x y)
    (loop_conditional (list x y) '() (- (len y) 1))
)
```

The test data was:

Input: X = 5 7 9 Y = 7 6 9
Output: 2 13 0

```
(cw '~x0~%'
    (ACL2_get_dat (car (main_conditional
                          (ACL2_mak_lst nil '(9 7 5))
                          (ACL2_mak_lst nil '(9 6 7))))))
(if (equal '(0 13 2)
    (ACL2_get_dat (car (main_conditional
                          (ACL2_mak_lst nil '(9 7 5))
                          (ACL2_mak_lst nil '(9 6 7))))))
  'passed conditional test'
  'failed conditional test')
```

Test Case 5: Inserting a constant at the beginning of a stream

This test case inserts the constant 5 at the first position of an integer input stream x. The length of the output stream is the same as the length of x.

The LUSTRE example was:

```
node sequence (x: int) returns (result: int);
let
   result = 5 -> pre (x);
tel
```

The ACL2 translation was:

```
(defun loop_sequence (inp res i)
  (if (zp i)
      res
      (let ((y (nth 0 inp))
            (res0 (nth 0 res)))
           (let* ((yi (ACL2_rem_lst y (- i 1)))
                  (res_curr (ACL2_pre_val yi 5)))
                 (loop_sequence
                    inp
                    (list (ACL2_add_val res0 nil res_curr))
                    (- i 1)))))))


(defun main_sequence (y)
    (loop_sequence (list y) '() (- (len y) 1))
)
```

The test data was:

Input: 1 2 6 8 9
Output: 5 1 2 6 8

```
(cw '~x0~%'
    (ACL2_get_dat (car (main_sequence (ACL2_mak_lst nil '(9 8 6 2 1))))))


(if (equal '(8 6 2 1 5)
    (ACL2_get_dat (car (main_sequence (ACL2_mak_lst nil '(9 8 6 2 1))))))
  'passed sequence test'
  'failed sequence test')
```

Test Case 6: Multiple equations as part of a single node

This test case validates use of equations. When multiple equations occur in a LUSTRE node, in the ACL2 translation, they have to be topologically sorted for an accurate translation.

The LUSTRE example was:

```
node equation (x:int; y: bool) returns (result: int);
var a: int; b: bool; c: int;
let
   result = if (b) then c + a else c - a;
   c = 5 -> pre(x);
   a = x + 2 - 1;
   b = (not y);
tel
```

The ACL2 translation was:

```
(defun loop_equation (inp tmp res i)
  (if (zp i)
      res
      (let ((x (nth 0 inp))
            (y (nth 1 inp))
            (a (nth 0 tmp))
            (b (nth 1 tmp))
            (c (nth 2 tmp))
            (res0 (nth 0 res)))
           (let* ((xi (ACL2_rem_lst x (- i 1)))
                  (yi (ACL2_rem_lst y (- i 1)))
                  (xi_curr (ACL2_cur_val xi))
                  (yi_curr (ACL2_cur_val yi))
                  (c_curr  (ACL2_pre_val xi 5))
                  (a_curr  (- (+ 2 xi_curr) 1))
                  (b_curr  (not yi_curr))
                  (res_curr (if b_curr
                                (+ c_curr a_curr)
                                (- c_curr a_curr))))
                 (loop_equation
                      inp
                      (list (ACL2_add_val a nil a_curr)
                            (ACL2_add_val b nil b_curr)
                            (ACL2_add_val c nil c_curr))
                      (list (ACL2_add_val res0 nil res_curr))
                      (- i 1)))))))


(defun main_equation (x y)
    (loop_equation (list x y) '() '() (- (len y) 1))
)
```

The test data was:

Input: X = 7 9 8 6 2 4 Y = T F T T F T
Output: -3 17 0 1 9 -3

```
(cw '~x0~%'
    (ACL2_get_dat (car (main_equation
       (ACL2_mak_lst nil '(4 2 6 8 9 7))
       (ACL2_mak_lst nil '(t nil t t nil t))))))

(if (equal '(-3 9 1 0 17 -3)
    (ACL2_get_dat (car (main_equation
       (ACL2_mak_lst nil '(4 2 6 8 9 7))
       (ACL2_mak_lst nil '(t nil t t nil t))))))
  'passed equation test one'
  'failed equation test one')
```

Input: X = 7 9 8 6 2 4 Y = F T F F T F
Output: 13 -3 18 15 3 7

```
(cw '~x0~%'
    (ACL2_get_dat (car (main_equation
       (ACL2_mak_lst nil '(4 2 6 8 9 7))
       (ACL2_mak_lst nil '(nil t nil nil t nil))))))

(if (equal '(7 3 15 18 -3 13)
    (ACL2_get_dat (car (main_equation
       (ACL2_mak_lst nil '(4 2 6 8 9 7))
       (ACL2 mak_lst nil '(nil t nil nil t nil))))))
  'passed equation test two'
  'failed equation test two')
```

**Test Set 3: Test Cases Involving Tuples:**

Test Case 7: Returning a constant output irrespective of the input with tuples

This test case takes an integer stream input y, ignores it and returns a constant integer stream which is the sum of values of a tuple.

The LUSTRE example was:

```
node constant_tuple (z: int) returns (res: int);
var x, y: int;
let
   (x, y) = (5, 7);
   res = x + y;
tel
```

The ACL2 translation was:

```
(defun loop_constant_tuple (tmp res i)
  (if (zp i)
      res
      (let (
            (x    (nth 0 tmp))
            (y    (nth 1 tmp))
            (res0 (nth 0 res)))
           (let* ((x_curr 5)
                  (y_curr 7)
                  (res_curr (+ x_curr y_curr)))
                 (loop_constant_tuple
                     (list (ACL2_add_val x nil x_curr)
                           (ACL2_add_val y nil y_curr))
                     (list (ACL2_add_val res0 nil res_curr))
                     (- i 1)))))))


(defun main_constant_tuple (y)
   (loop_constant_tuple '() '() (- (len y) 1))
)
```

The test data was:

Output: 12 12 12 12

```
(cw '~x0~%'
    (ACL2_get_dat (car (main_constant_tuple (ACL2_mak_lst nil '(1 1 1
1))))))
(if (equal '(12 12 12 12)
    (ACL2_get_dat (car (main_constant_tuple (ACL2_mak_lst nil '(1 1 1
1))))))
  'passed constant_tuple test'
  'failed constant_tuple test')
```

Test Case 8: Execution of an if-then-else statement with tuples

This test case checks the implementation of an if-then-else statement with tuples.

The LUSTRE example was:

```
node conditional_tuple (x: int; y: int) returns (r1, r2: int);
let
    (r1, r2) = if (x > y)
                  then (x + y, 1)
                  else (y - x, 2);
tel
```

The ACL2 translation was:

```
(defun loop_conditional_tuple (inp res i)
  (if (zp i)
      res
      (let (
            (x    (nth 0 inp))
            (y    (nth 1 inp))
            (r1   (nth 0 res))
            (r2   (nth 1 res)))
           (let* (
                  (xi     (ACL2_rem_lst x (- i 1)))
                  (yi     (ACL2_rem_lst y (- i 1)))
                  (x_curr (ACL2_cur_val xi))
                  (y_curr (ACL2 cur val yi))
                  (r1_curr (if  (> x_curr y_curr)
                                (+ x_curr y_curr)
                                (- y_curr x_curr)))
                  (r2_curr (if  (> x_curr y_curr)
                                1
                                2)))
                 (loop_conditional_tuple
                     inp
                     (list
                         (ACL2_add_val r1 nil r1_curr)
                         (ACL2_add_val r2 nil r2_curr))
                     (- i 1)))))))


(defun main_conditional_tuple (x y)
   (loop_conditional_tuple (list x y) '() (- (len y) 1))
)
```

The test data was:

Input: X = 2 5 2 5  Y = 6 3 9 8

Output: (4 8 7 3) (2 1 2 2)

```
(cw '~x0~%'
    (ACL2_get_dat (car (main_conditional_tuple
       (ACL2_mak_lst nil '(2 5 2 5))
       (ACL2_mak_lst nil '(6 3 9 8))))))


(let* ((tmp  (main_conditional_tuple
                (ACL2_mak_lst nil '(2 5 2 5))
                (ACL2_mak_lst nil '(6 3 9 8))))
       (x    (nth 0 tmp))
       (y    (nth 1 tmp)))
      (if (and (equal '(4 8 7 3) (ACL2_get_dat x))
               (equal '(2 1 2 2) (ACL2_get_dat y)))
          'passed conditional_tuple test'
          'failed conditional_tuple test'))
```

Test Case 9: Multiple equations as part of a single node with tuples

This test case validates use of equations with tuples. When multiple equations occur in a LUSTRE node, in the ACL2 translation, they have to be topologically sorted for an accurate translation.

The LUSTRE example was:

```
node equation_tuple (x: int; y: bool) returns (r1: int);
var a: int; b: bool; c: int;
let
    r1 = if (b) then c + a else c - a;

    (c, a, b) = (5 -> pre(x), x + 2 - 1, (not y));
tel
```

The ACL2 translation was:

```
(defun loop_equation_tuple (inp res i)
  (if (zp i)
      res
      (let (
            (x    (nth 0 inp))
            (y    (nth 1 inp))
            (r1   (nth 0 res)))
           (let* (
                  (xi     (ACL2_rem_lst x (- i 1)))
                  (yi     (ACL2_rem_lst y (- i 1)))
                  (x_curr (ACL2_cur_val xi))
                  (y_curr (ACL2_cur_val yi))
                  (c_curr (ACL2_pre_val xi 5))
                  (a_curr (- (+ x_curr 2) 1))
                  (b_curr (not y_curr))
                  (r1_curr (if b_curr
                               (+ c_curr a_curr)
                               (- c curr a_curr))))
                 (loop_equation_tuple
                     inp
                     (list (ACL2_add_val r1 nil r1_curr))
                     (- i 1)))))))


(defun main_equation_tuple (x y)
   (loop_equation_tuple (list x y) '() (- (len y) 1))
)
```

The test data was:

Input: X = 3 7 4 99 5 8 3  Y = T T F T T F T

Output: 3 -4 104 -95 2 12 1

```
(cw '~x0~%'
    (ACL2_get_dat (car (main_equation_tuple
       (ACL2_mak_lst nil '(3 7 4 99 5 8 3))
       (ACL2_mak_lst nil '(1 1 nil 1 1 nil 1))))))


(if (equal '(3 -4 104 -95 2 12 1)
           (ACL2_get_dat (car (main_equation_tuple
               (ACL2_mak_lst nil '(3 7 4 99 5 8 3))
               (ACL2_mak_lst nil '(1 1 nil 1 1 nil 1))))))
         'passed equation_tuple test'
         'failed equation_tuple test'))
```

**Test Set 4: Test Case With Multiple Nodes:**

Test Case 10: Test case with 2 nodes:

This is a simple test that incorporates 2 nodes.

The LUSTRE example was:

```
node two_functions (z: int) returns (res: int);
var t1, t2, t3, t4: int;
let
   (t1, t2) = subtest(z);
   t3 = 5 -> pre(t1);
   t4 = 6 -> pre(t2);
   res = t3 + t4;
tel

node subtest(z: int) returns (z1, z2: int);
let
   z1 = z + 1;
   z2 = z + 2;
tel
```

The ACL2 translation was:

```
(defun loop_up_main_subtest (inp res i)
  (if (zp i)
      res
      (let ((z (nth 0 inp))
            (z1 (nth 0 res))
            (z2 (nth 1 res)))
           (let* ((z_curr (ACL2_cur_val (ACL2_rem_lst z (- i 1))))
                  (z1_curr (+ 1 z_curr))
                  (z2_curr (+ 2 z_curr)))
                 (loop_up_main_subtest
                     inp
                     (list
                        (ACL2_add_val z1 nil z1_curr)
                        (ACL2_add_val z2 nil z2_curr))
                     (- i 1)))))))


(defun main_subtest (y)
   (loop_up_main_subtest (list y) '() (- (len y) 1))
)


(defun loop_up_main_two_functions (inp res i)
  (if (zp i)
      res
      (let ((y (nth 0 inp))
```

```
            (res0 (nth 0 res)))
           (let* (
                  (yi (ACL2_rem_lst y (- i 1)))
                  (tuple_1 (main_subtest yi))
                  (t1 (nth 0 tuple_1))
                  (t2 (nth 1 tuple_1))
                  (t3_curr (ACL2_pre_val t1 5))
                  (t4_curr (ACL2_pre_val t2 6))
                  (res curr (+ t3_curr t4_curr)))
                 (loop_up_main_two_functions
                     inp
                     (list (ACL2_add_val res0 nil res_curr))
                     (- i 1))))))

(defun main_two_functions (z)
   (loop_up_main_two_functions (list z) '() (- (len z) 1))
)
```

The test data was:

Input: 1 2 3 4 5

Output: 11 5 7 9 11

```
(cw '~x0~%' (car (main_two_functions
    (ACL2_mak_lst nil '(5 4 3 2 1)))))

(if (equal '(11 9 7 5 11) (ACL2_get_dat (car (main_two_functions
    (ACL2_mak_lst nil '(5 4 3 2 1))))))
  'passed two_functions test'
  'failed two_functions test')
```

**Test Set 5: Test Cases Involving Clocks:**

Test Case 11: Basic test case for 'when' and 'current'

This is a simple test that demonstrates the use of 'when' and 'current'.

The LUSTRE example was:

```
node whenCurrent (y: int; b: bool) returns (x: int);
let
   x = current (y when b);
tel
```

The ACL2 translation was:

```
(defun loop_whenCurrent (inp res i)
  (if (zp i)
      res
      (let ((y (nth 0 inp))
            (b (nth 1 inp)))
           (let* ((yi (ACL2_rem_lst y (- i 1)))
                  (bi (ACL2_rem_lst b (- i 1)))
                  (xi (ACL2_current (ACL2_when yi bi))))
                 (loop_whenCurrent inp
                           (list xi)
                           (- i 1))))))


(defun main_whenCurrent (y b)
    (loop_whenCurrent (list y b) '() (- (len y) 2))
)
```

The test data was:

Input: Y = 1 2 3 4 5 6 7  B = T F F  T F F T
Output: 1 1 1 4 4 4 7

```
(cw '~x0~%'
     (ACL2_get_dat (car (main_whenCurrent
      (ACL2_mak_lst '() '(7 6 5 4 3 2 1))
       (ACL2_mak_lst '() '(1 nil nil 1 nil nil 1))
      ))))


(if (equal '(7 4 4 4 1 1 1)
    (ACL2_get_dat (car (main_whenCurrent
      (ACL2_mak_lst '() '(7 6 5 4 3 2 1))
      (ACL2_mak_lst '() '(1 nil nil 1 nil nil 1))))))
  'passed whenCurrent test'
  'failed whenCurrent test')
```

Test Case 12: Second test case for 'when' and 'current' – 2 nodes

This is the second test that demonstrates the use of 'when' and 'current'. It comprises of 2 nodes.

The LUSTRE example was:

```
node whenCurrent2(y, z: int; b: bool; c: bool) returns (x: int);
var t: int;
let
   t = current ((y when b) + fun_a ((y, z) when b));
   x = current (t when c);

tel


node fun_a (x, y: int) returns (r: int);
let
  r = x*y;
tel
```

The ACL2 translation was:

```
(defun loop_fun_a (inp res i)
  (if (zp i)
      (list (cons (car (nth 0 inp)) (cdr (nth 0 res))))
      (let ((x (nth 0 inp))
            (y (nth 1 inp))
            (r (nth 0 res)))
           (let* ((xi (ACL2_rem_lst x (- i 1)))
                  (yi (ACL2_rem_lst y (- i 1)))
                  (x_curr (ACL2_cur_val xi))
                  (y_curr (ACL2_cur_val yi))
                  (r_curr (* x_curr y_curr)))
                 (loop_fun_a
                     inp
                     (list (ACL2_add_val r nil r_curr))
                     (- i 1)))))))

(defun fun_a (x y)
   (loop_fun_a (list x y) '() (- (len x) 1)))


(defun loop_plus (inp res i)
   (if (zp i)
      (list (cons (car (nth 0 inp)) (cdr (nth 0 res))))
      (let ((x (nth 0 inp))
            (y (nth 1 inp))
```

```
            (res0 (nth 0 res)))
           (let* ((xi (ACL2_rem_lst x (- i 1)))
                  (yi (ACL2_rem_lst y (- i 1)))
                  (xi_curr (ACL2_cur_val xi))
                  (yi_curr (ACL2_cur_val yi))
                  (res_curr (+ xi_curr yi_curr)))
              (loop_plus
                 inp
                 (list (ACL2_add_val res0 nil res_curr))
                 (- i 1))))))


(defun ACL2_plus (t1 t2)
  (loop_plus (list t1 t2) '() (- (len t1) 1)))

(defun loop_whenCurrent2 (inp res i)
  (if (zp i)
      res
      (let ((y (nth 0 inp))
            (z (nth 1 inp))
            (b (nth 2 inp))
            (c (nth 3 inp)))
           (let* ((yi (ACL2_rem_lst y (- i 1)))
                  (zi (ACL2_rem_lst z (- i 1)))
                  (bi (ACL2_rem_lst b (- i 1)))
                  (ci (ACL2_rem_lst c (- i 1)))
                  (tmp1 (ACL2_when yi bi))
                  (tmp2 (ACL2_when zi bi))
                  (tmp3 (car (fun_a tmp1 tmp2)))
                  (tmp4 (ACL2_plus tmp1 tmp3))
                  (t1 (ACL2_current (car tmp4)))
                  (x (ACL2_current (ACL2_when t1 ci))))
                  (loop_whenCurrent2
                  inp
                  (list x)
                  (- i 1))))))


(defun main_whenCurrent2 (y z b c)
    (loop_whenCurrent2 (list y z b c) '() (- (len y) 2))
)
```

The test data was:

Input: y 5 7 9 1 6 4 3 7 z 3 4 2 9 1 7 1 6
b 1 0 0 1 0 0 1 0 c 0 0 1 0 0 1 0

Output: nil nil 20 20 20 10 10 6

```
(cw '~x0~%'
       (ACL2_get_dat (car (main_whenCurrent2
         (ACL2_mak_lst '() '(7 3 4 6 1 9 7 5))
         (ACL2_mak_lst '() '(6 1 7 1 9 2 4 3))
         (ACL2_mak_lst '() '(nil 1 nil nil 1 nil nil 1))
         (ACL2_mak_lst '() '(1 nil 1 nil nil 1 nil nil))
         ))))
```

```
(if (equal '(6 10 10 20 20 20 nil nil)
     (ACL2_get_dat (car (main_whenCurrent2
          (ACL2_mak_lst '() '(7 3 4 6 1 9 7 5))
          (ACL2_mak_lst '() '(6 1 7 1 9 2 4 3))
          (ACL2_mak_lst '() '(nil 1 nil nil 1 nil nil 1))
          (ACL2_mak_lst '() '(1 nil 1 nil nil 1 nil nil))))))
   'passed whenCurrent2 test'
   'failed whenCurrent2 test')
```

Test Case 13: Third test case for 'when' and 'current' – 2 nodes and 'pre'

This is the thrid test that demonstrates the use of 'when' and 'current'. It comprises of 2 nodes and 'pre'.

The LUSTRE example was:

```
node testWhen3(y, z: int; b: bool) returns (r1, r2: int);
let
   r1 = current (fun_a((y, z) when b));
   r2 = current (fun_a(y, z)  when b);

tel

node fun_a (x, y: int) returns (r: int);
let
   r =  (5 -> pre (x)) + (10 -> pre(y));
tel
```

The ACL2 translation was:

```
(defun loop_fun_a (inp res i)
  (if (zp i)
      (list (cons (car (nth 0 inp)) (cdr (nth 0 res))))
      (let ((x (nth 0 inp))
            (y (nth 1 inp))
            (r (nth 0 res)))
           (let* ((xi (ACL2_rem_lst x (- i 1)))
                  (yi (ACL2_rem_lst y (- i 1)))
                  (r_curr (+
                     (ACL2_pre_val xi 5)
                     (ACL2_pre_val yi 10))))
                 (loop_fun_a
                     inp
                     (list (ACL2_add_val r nil r_curr))
                     (- i 1)))))))

(defun fun_a (x y)
   (loop_fun_a (list x y) '() (- (len x) 1)))


(defun loop_whenCurrent3 (inp res i)
  (if (zp i)
      res
      (let ((y (nth 0 inp))
            (z (nth 1 inp))
            (b (nth 2 inp)))
           (let* ((yi (ACL2_rem_lst y (- i 1)))
                  (zi (ACL2_rem_lst z (- i 1)))
                  (bi (ACL2_rem_lst b (- i 1)))
```

```
                    (tmp1 (ACL2_when yi bi))
                    (tmp2 (ACL2_when zi bi))
                    (tmp3 (car (fun_a tmp1 tmp2)))
                    (r1   (ACL2_current tmp3))
                    (tmp4 (car (fun_a yi zi)))
                    (r2   (ACL2_current
                             (ACL2_when tmp4 bi))))
                   (loop_whenCurrent3
                   inp
                   (list r1 r2)
                   (- i 1))))))

;;   r1 = current (fun_a((y, z) when b));
;;   r2 = current (fun_a(y, z)  when b);

(defun main_whenCurrent3 (y z b)
    (loop_whenCurrent3 (list y z b) '() (- (len y) 2))
)
```

The test data was:

Input: y 1 2 3 4 5  z 9 10 11 12 13  b 1 0 1 0 1
Output: r1 15 15 10 10 14  r2 15 15 12 12 16

```
(cw '~x0~%'
       (main_whenCurrent3
         (ACL2_mak_lst '() '(5 4 3 2 1))
         (ACL2_mak_lst '() '(13 12 11 10 9))
         (ACL2_mak_lst '() '(1 nil 1 nil 1))
         ))



(let* ((tmp  (main_whenCurrent3
         (ACL2_mak_lst '() '(5 4 3 2 1))
         (ACL2_mak_lst '() '(13 12 11 10 9))
         (ACL2_mak_lst '() '(1 nil 1 nil 1))))
       (x    (nth 0 tmp))
       (y    (nth 1 tmp)))
      (if (and (equal '(14 10 10 15 15) (ACL2_get_dat x))
               (equal '(16 12 12 15 15) (ACL2_get_dat y)))
          'passed whenCurrent3 test'
          'failed whenCurrent3 test'))
```