A methodology for proving control systems with Lustre and PVS*†

S.Bensalem, P.Caspi, C.Parent-Vigouroux, C.Dumas
VERIMAG
name@imag.fr

Abstract

In this paper, we intend to show how to use the synchronous data-flow language Lustre, combined with the PVS proof system in deriving provably-correct (distributed) control programs. We hopefully illustrate, based on a railway emergency braking system example, the features of our approach — asynchronous periodic programs with nearly the same period, communicating by sampling — equational reasoning which leaves to the Lustre compiler the task of scheduling computations — no distinction between control programs and physical environments which are sampled in the same way. This allows us to provide "elementary" proofs based on difference equations instead of differential ones which require more involved PVS formalization.

1 Introduction

Control systems form an important class of critical computer systems: it is in this domain that some of the most critical applications can be found, for instance in civil aircrafts, ground transportation, nuclear power etc. Thus, a lot of activity has been devoted to ensuring and improving hardware and software reliability. Concerning software, fault avoidance has always been, beside fault tolerance, an important issue and program deductive proofs can be, in complement with other methods, a tentative answer to the problem. Proving software is a broad subject which has drawn the attention of many researchers for a long time and tools, languages, and methods become widely available for handling the problem in general. We want to address here the problem of using these tools within the context of critical control systems. This, in our opinion, requires taking into account the following features of the domain:

*This work has been partially supported by Esprit Projects SYRF and CRISYS, and INRIA action PRESYSA

†This paper appeared in the proceedings of the seventh Working Conference on Dependable Computing for Critical Applications (DCCA7), San Jose, January 1999.

Data-flow: Control engineers use to think (analyze, specify) of their systems in terms of data-flow. For instance, they use tools such as Matlab and MatrixX to build and simulate systems. Translating these specifications into imperative programs can be seen as a boring and error-prone task. On the contrary, it may be expected that the use of synchronous data-flow languages is likely to alleviate this task and provide for safer programs.

Distribution: Beside the fact that most control systems are distributed, there is an intrinsic factor of distribution based on the division between the system and its environment. In general for control systems, it is not possible to even express (not to speak of proving) properties of the system without taking into account its environment (closed loop systems). Thus distribution has to be carefully taken into account, and some methodology is needed for that.

Hybridity: Furthermore, this environment is in most cases continuous, modeled with differential equations, for instance, while computers have discrete time behaviors. This is an important issue that must be carefully discussed and taken into account.

1.1 Paper content

In this paper we shall try to address these issues through a toy example, consisting of an emergency brake system for railways. The data-flow aspect will be taken into account by using the Lustre [9] synchronous data-flow language for describing and programming systems and by using the PVS [18] verification system for handling proofs of Lustre programs. Distribution will be handled here by expressing *both* the control program, the environment, and communication between both within the *same* Lustre-PVS framework, thanks to the sampling and holding primitives of Lustre. This calls for some methodological precaution, consisting of carefully distinguishing between environment variables and the way they are perceived by the control program. This raises the question: since Lustre does not allow for continuous time systems, how to take into account the hybrid aspect of the example? The proposed answer is quite simple: instead of building a complex theory of continuity and mathematical analysis in PVS, we propose here to simply take care of it through a careful handling of integration errors (in the same way as engineers compute bounds on errors).

This will be presented as follows: first we shall present Lustre, PVS and the principles of a translation tool from Lustre to PVS. Then we present in Lustre the toy example and the PVS proof of its principle. Finally, we introduce the distributed nature of the example and take care of the environment.

1.2 Related issues

On the one hand, many aspects of control systems are shared with hardware systems and protocols, and a long lasting research has been devoted to them, within the verification community. However, most associated tools are based on (automatic) algorithmic methods, while control is mostly based on numerical properties which can hardly be automatically handled for reasons of complexity and decidability. For instance the Lustre language has an associated model-checker, Lesar which has been successfully applied to hardware and simple control problems [10]. However, we have checked that it cannot cope with the problems we address here.

On the other hand, deductive tools like PVS [18] and Coq [6] provide very general frameworks that need being specialized to some particular domain.

In between we find methods well adapted to sequential programming like B and VDM [1, 12]. These methods can be adapted to the problems we want to address, but there is some price to pay for it, because data-flow programming is not natural here: in [1], Abrial describes and handles a control problem, the celebrated steam boiler. Yet a large part of the control program derivation consists of sequencing the program and proving it, an operation which is for instance automatically performed by the Lustre compiler.

We also find specific proof tools adapted to reactive systems like the TLA approach [15]. This method is more general than our Lustre-PVS one, in the sense that it can handle both synchronous and asynchronous systems and could as well support the proposed methodology.

Finally, we find so-called hybrid system methods and tools. For instance a very similar problem is handled in PVS by N.Lynch and co-authors [7]. However we differ from this approach because we think continuous time is uselessly complex and can be avoided by sampling both the system and its environment. This is clearly an opinion that somewhat contradicts current research trends on hybrid systems. Yet, using computers for controlling physical systems is a long lasting activity which goes back to the early eighties [14]. According to control theory, there are two possible approaches: either design a continuous controller and then sample it, or sample the process to be controlled and design a sampled controller based on sampled control system theory. We follow here the later approach.

2 The Lustre-PVS approach

Synchronous data-flow programming has become a popular approach, for the quite obvious reason that automatic control engineers use to analyze and specify their systems in terms of signals flowing through networks of operators (block diagrams). Thus, synchronous data-flow languages allow them to directly translate their specifications into code without undertaking a manual, error prone, coding phase. Several real world achievements have been obtained in this way, among which we can cite the "fly-by-wire" system of Airbus aircrafts [4], the monitoring system of FRAMATOME nuclear plants, designed by SCHNEIDER ELECTRIC [2], and the interlocking system of the Hong Kong subway designed by CS TRANSPORT. Of these examples, the last two ones have been based on the Lustre data-flow synchronous language [9].

2.1 The Lustre language

In this presentation, we focus on semantic aspects, perhaps at the expense of syntactic details, the more so as our main interest is in the translation to PVS.

2.1.1 Basic concepts Basically, 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 thanks to 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 $[t, t, t \ldots]$
- variables denote streams,

`x`	x_0	x_1	x_2	$\ldots$

- Usual operators operate pointwise over streams; for instance `1 + 2` is the stream $[3, 3, \ldots]$. Similarly,

`c`	t	f	t	$\ldots$
`x`	x_0	x_1	x_2	$\ldots$
`y`	y_0	y_1	y_2	$\ldots$
`if c then x else y`	x_0	y_1	x_2	$\ldots$

- A unit delay operator `fby` "followed-by" (actually `->pre` in Lustre) can be represented by the diagram:

`x`	x_0	x_1	x_2	$\ldots$
`y`	y_0	y_1	y_2	$\ldots$
`x fby y`	x_0	y_0	y_1	$\ldots$

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	$\ldots$
`y`	x_0	$x_0 + x_1$	$x_0 + x_1 + x_2$	$\ldots$
`0 fby y`	0	x_0	$x_0 + x_1$	$\ldots$

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 concepts

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

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

Node parameters are typed and the compiler performs 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.

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 in the synchronous nature of Lustre.

Assertions: Always "true" boolean expressions can be asserted for several purposes, for instance for expressing non independent input properties; for instance:

```
assert (c or (true fby c));
```

says that `c` will not stay "false" for more than one time unit.

2.2 Lustre in PVS

In this section, we first give a brief description of the PVS system and then we show how Lustre semantics can be specified using PVS.

2.2.1 PVS Features PVS (Prototype Verification System) [18] is an environment for constructing specifications and verifications that has been developed at SRI International's Computer Science Laboratory. In comparison to other widely used verification systems, such as Coq [6], HOL [8] and the Boyer-Moore prover [3], the distinguishing characteristic of PVS is a coherent realization, combining within a single system:

- a highly expressive specification language which is based on higher-order logic with a parser, and an expressive type system. The type-checker checks for simple type correctness and generates proof obligations. A specification in PVS is organized in theories, which may depend on other theories via an importing mechanism. In particular, any theory may import from the set of built-in theories. A theory consists of a sequence of declarations, which provide names for types, constants, variables, axioms, and formulas. The language supports modularity and reuse by means of parameterized theories.
- an interactive proof checker consisting of a higher-level collection of inference steps (written in a strategy language) that can be used to reduce the proof goal to simpler subgoals that can be proved automatically by primitive rule steps of the prover. The primitive proof steps for constructing proofs involve efficient decision procedures for equality and linear arithmetic, automatic rewriting, and BDD-based propositional simplifications.

2.2.2 Lustre in PVS We now focus on the specification of the Lustre constructs in PVS. In doing this, we face the problem that, as we have seen at section 2.1.1, streams can be recursively defined in Lustre programs. But, unfortunately, these "non-terminating" recursive definitions are not allowed in PVS. All definitions in PVS are total [17], and form a conservative extension. In order to guarantee these

```
lustre [t:TYPE] : THEORY

  BEGIN
  x      : VAR t
  i      : VAR nat
  c      : VAR sequence[bool]
  xs, ys : VAR sequence[t]

  const(x)(i) : t = x
  CONVERSION const

  fby(xs,ys)(i) : t = IF i=0 THEN xs(0) ELSE ys(i-1) ENDIF

  ifl(c,xs,ys)(i) : t = IF c(i) THEN xs(i) ELSE ys(i) ENDIF

  END lustre
```

Table 1: Lustre theory

conditions, a function named `MEASURE` is required. This function is used to show that the definition terminates, by generating a proof obligation consisting of proving that the `MEASURE` function decreases with each call. There exist at least two ways of overcoming this problem:

- the first solution uses a functional style and views streams as a co-inductive type [19, 16, 11],
- the second one consists simply of viewing streams as infinite sequences. In this framework, a constant stream `v` of type `t` would be translated in PVS as:

  ```
  const(v:t) : sequence[t] = LAMBDA(i) : v
  ```

  that is to say, the function which associates with any natural number `i` the value `v`.

2.2.3 Specification of Lustre primitives The parameterized theory `lustre`, displayed at table 1 implements the second solution. `ifl` is the extension of the current conditional over sequences. The parameter `t` is treated as a fixed uninterpreted type. When the theory lustre is used by another theory it must be instantiated.

The parameterized theory `lustre_extend`, displayed at table 2 allows us to extend pointwise operators over sequences.

The specification of `current` and `when` are more complicated than the previous ones. We need an intermediate function `count` : intuitively, `count` can be seen as a function on natural numbers which counts the number of `c(i) = true` for

```
lustre_extend [t1,t2:TYPE] : THEORY

  BEGIN

  i     : VAR nat
  xs,ys : VAR sequence[t1]
  f     : VAR [t1->t2]
  g     : VAR [t1,t1->t2]

  map(f)(xs)(i) : t2 = f(xs(i))

  map2(g)(xs,ys)(i) : t2 = g(xs(i),ys(i))

  CONVERSION lustre_extend.map, map2

  END lustre_extend
```

Table 2: Map theories

$0 \leq i \leq n$. Furthermore, we need to restrict to clocks having an infinite number of occurrences, so as to keep infinite sequences when filtering. The corresponding theory, inspired from [5] is shown at table 3.

2.2.4 The feedback loop problem

Yet this still doesn't solve the problem of Lustre recursive stream definitions; for instance we would like to write:

```
integr( x : sequence[int] ) : sequence[int] = y
  WHERE y =  fby(0, y) +  x
```

which is not allowed. We propose here to overcome the problem by writing instead the input/output predicate:

```
integr(x,y:sequence[real]) : THEORY
  BEGIN

  y_ax : LEMMA y =  fby(0, y) +  x

  END integr
```

However, this is not strictly equivalent, in the sense that we may now have either several or no output sequence corresponding to one input sequence. This solution works because we only aim at translating valid Lustre programs for which the compiler has checked for functionality and deadlock-freedom.

In this context, this solution looks acceptable and then yields valuable benefits:

- it is easy to read and user-friendly: input-output predicates encapsulate equations very close to the original Lustre equations,

```
exten2_lustre [ t : TYPE] : THEORY
  BEGIN
    IMPORTING lustre, lustre_extend, bounds

    clock : TYPE = {c:sequence[bool] |  (NOT c(0)) AND
                   FORALL (n: nat):
                     EXISTS (m: nat): m >= n and c(m)}

    n             : VAR nat
    c             : VAR clock
    xs, ys        : VAR sequence[t]

    count(c) (n): recursive nat =
     IF c(n) THEN 1 ELSE 0 ENDIF +
     IF n=0 THEN 0 ELSE count(c)(n-1) ENDIF
     MEASURE n

    current (c,xs) (ys) (n) : t =
      LET m = count(c)(n)
      IN IF m = 0 THEN xs(0) ELSE ys(m-1) ENDIF

    index(c)(n) : nat = glb {m:nat | count(c)(m) >= n}

    lwhen(c,xs)(n) : t = xs(index(c)(n+1))

  END exten2_lustre
```

Table 3: Extensions

- it sometimes yields very simple proofs: for instance, the PVS proof of

  ```
  map2(+)(const(1), const(2)) = const(3)
  ```

  is automatic and doesn't use any induction at all.

- a disadvantage of this approach is that, while functional composition is straightforward, composing predicates requires creating intermediate variables unknown to the programmer. We are currently working on an alternative solution based on the theory of Kahn networks [13].

3 A toy railway control problem

This section demonstrates our approach using a simple but non-trivial example taken from the automatic control of a railway systems. Each train on the track has a computerized controller which periodically receives some information about its speed, given by some sensor, and the position of the train in front of it, broadcasted by some transmission device. Based on this information the controller decides whether or not to actuate an emergency brake in order to avoid colliding with the preceding train.

3.1 Specification

In this specification, all variables are Lustre variables and denote streams. Our specification is made, first, of assumptions about the physical world. For this, we consider the train 1 that is behind:

- Its acceleration, `a`, is, if the controller actuates the brake then `-afu`, where `afu` is the (positive) deceleration of the emergency brake, otherwise `am`, some unknown stream, yet not larger than a maximal acceleration constant `amax`:
  ```
  const afu, amax : real;
  a = if fu then -afu else am;
  assert( (afu > 0 ) and (am < amax) );
  ```

- Trains are not supposed to move backward and are assumed to be initially stopped.
  ```
  const dt :real;
  assert dt>0;
  vv = (0 fby v) + a*dt;
  v = if vv>=0 then vv else 0;
  ```
  where `dt` is the time period at which the system runs.

- The distance between trains 2 and 1, `d`, is the difference between the position of train 2, `x2` and the integral of the speed of train 1 `v`. A positive initial distance is assumed, and train 2 does not move backward:

```
assert ((x2   >= 0) fby true)
          and x2 - (x2 fby x2) >= 0);
x = (0 fby x) + v*dt;
d = x2 - x;
```

Given this environment a specification could be:

Find a controller law for `fu` *such that trains never collide:* `d > 0`

3.2 A controller proposal

`fu` is a boolean stream which says whether the train has to brake or not. To synthesize it, at each time, the worst situation has to be considered: train 2 stops (its speed is 0) and train 1 has the maximal acceleration `amax`. If, when in this situation, the train waits the next instant for braking and this results in a collision, then it must begin braking now.

```
fu = df((x2 fby (d - v*dt)) - amax*dt*dt,
        (0 fby v) + amax*dt
       )< 0 ;
```

where `df` is the final distance function: `df(d,v)` is the distance between the current train and the preceding one that will be reached starting from distance `d` and speed `v` if the later doesn't move and if the former continuously brakes with deceleration `-a`:

$floor(\frac{v}{a*dt}) = u$ is the number of steps up to which the speed will get zero. Thus,

$$df(d, v) = d - \sum_{i=1}^{u}(v.dt - \sum_{j=1}^{i} a.dt^2) = d - uv.dt + \frac{u(u+1)}{2}a.dt^2$$

3.3 Controller correctness

In order to prove the controller correctness, we translate the specification into PVS. The translation is shown at table 4. Then, we prove that the two trains never collide.

```
correctness : THEOREM FORALL (n:nat): prop(n)
```

In doing this, we need the following steps:

1. speed is always positive,

   ```
   vpos : THEOREM FORALL (n:nat): v(n) >= 0
   ```

2. a positive final distance is a system invariant

   ```
   inv : THEOREM FORALL (n:nat): df(d(n),v(n)) >=0
   ```

```
train: THEORY [am,x2,v,d:sequence[real],prop:sequence[bool]]
 BEGIN
   IMPORTING lustre, lustre_extend

    a  : sequence[real]
    fu : sequence[bool]

    amax,afu,dt : real

    a_ax : AXIOM a = ifl(fu,-afu,am)

    v_ax : AXIOM v = let vv = fby(0,v) + a*dt
                     in  ifl(vv >= 0 ,vv, 0)

    x_ax : AXIOM x = fby(0,x) + v*dt

    d_ax : AXIOM d = x2 - x

    fu_ax: AXIOM fu =
       df( fby (x2, d-v*dt)- amax*dt*dt,
           fby (0, v)+ amax*dt
         )< 0

    assert_ax : AXIOM
       afu > 0 AND am < amax AND dt > 0
          AND fby((x2  >= 0), true)
             AND x2 - fby(x2,x2) >= 0

    prop_ax : AXIOM prop = d >=0

  END train
```

Table 4: Train theory

3. as a consequence, the trains will not collide.

The proofs are mainly based on induction over naturals and on algebraic properties of equalities and inequalities. Note two important lemmas that are proved meanwhile and that will be very useful in the sequel:

- the final distance function is an increasing function of `d` and a decreasing function of `v`:

```
mondf : LEMMA
  FORALL (d1,d2,v1,v2:real):
    d1<= d2 AND v2 >=0 AND v2 <= v1
      => df(d1,v1)  <= df(d2,v2)
```

- when braking, the final distance function remains constant:

```
dminus : LEMMA
  FORALL (d,v:real): v>=0 AND a >= 0
    =>  df(d,v) = df(d-(v + a*dt)*dt, v-a*dt)
```

4 Implementation

Up to now, the proposed system is an abstract one. In order to make it realistic, we must split it into parts:

- the train, which receives from the controller an acceleration and "computes" a speed and a distance,
- the controller which reads a speed , receives a position from the train ahead, and computes an acceleration,
- and the controller of the train ahead which communicates its position.

4.1 The distributed system

4.1.1 The train The train is merely the preceding train, but for the acceleration and control law:

```
const  dt :real;

node train(a, x2:real) returns (v, x, d:real);
var vv:real;
let
  vv = (0 fby v) + a*dt ;
  v = if vv>=0 then vv else 0;
  x = (0 fby x)  + v*dt;
  d = x2 - x;
  assert   ((x2 >= 0) fby true)
           and  (x2 - (x2 fby x2) >= 0);
tel
```

4.1.2 The train controller It performs the other computations. Yet, it does not receive the distance and has to recompute it. Thus, we shall have in the system two copies of the distance: the "real" one, computed by `train` and the computed one, computed by `controller`. Also note that the controller issues its position toward the train behind it.

```
const afu, amax: real;

node control (am, v, x2 :real) returns (a, x  :real);
var d:real; fu: bool;
let
  assert am < amax;
  x = (0 fby x) + v*dt;
  d = x2 - x;
  fu  = df((x2 fby (d - v*dt)) - amax*dt*dt,
          (0 fby v) + amax*dt
         )< 0;
  a   = if fu then -afu else am;
tel
```

4.2 Subsystem communication

Now, those subsystems need to communicate. The idea here is that both the physical world `train` and train controllers run periodically with the same period `dt` but *asynchronously* and communicate by sampling. This is realistic concerning communications between computers and corresponds to actual popular implementations in the control field: networks of periodic computers communicating by shared memory, periodic field busses, serial lines, or even radio. It may be less realistic when it comes to the relations between a computer and a physical environment. However, in many cases, it looks convenient because most often controllers need to simulate the environment, so as to take correct decisions [1]. Thus, the period of the controller has to be small enough in order to get an accurate simulation. Simulating the actual environment with the same period will then also be accurate, and the only assumption we need is the asynchronous sampling communication between them.

This can be modeled as follows: Each process `p` runs on its own clock `cp` . A value `v` issued from `p` is visible from outside through its past current value:

```
current(cp,v0)(v)
```

where `v0` is an initial value known by everybody. When another process `p'` wants to read it, it just samples it according to its own clock `cp'`.

Furthermore, we will need, in the proofs, to express the fact that those clocks, though independent, share the same period. This will be done thanks to an assertion stating that there cannot be more than two ticks of clock 1 between two ticks of clock 2 and conversely. Let `same_period` be that property. It can be strengthened to account for any clock drift by:

[1] This is the case of our controller, which has to integrate the speed for computing the distance.

```
const  dt, afu, amax : real; drinv:int;
node system(am1,am2,x3,xp3:real;c1,c2,ct:bool)
     returns (prop:bool)

var v1,v2,x1,x2,xp1,xp2,a1,a2,d1,d2:real;
let
  v1,x1,d1 = current(ct,(0,0,0))train((a1,x2)when ct);
  v2,x2,d2 = current(ct,(0,0,0))train((a2,x3)when ct);
  a1,xp1 = current(c1,(0,0))control((am1,v1,xp2) when c1);
  a2,xp2 = current(c2,(0,0))control((am2,v2,xp3) when c2);
  assert     same_period(ct,c1,drinv)
         and same_period(ct,c2,drinv)
         and same_period(c1,c2,drinv);
  assert dt > 0 and afu > 0 and amax > 0;
  prop = d1 >= 0;
tel
```

Table 5: The whole system

```
c1,c2    : VAR clock
driftinv : VAR nat;

drift_def(c1,c2,driftinv) : bool =
  FORALL (n,m :nat): m <= driftinv
     =>
  index(c1)(n+m+1)>index(c2)(count(c2)(index(c1)(n))+m)

same_period(c1,c2,driftinv) : bool =
  drift_def(c1,c2,driftinv) AND drift_def(c2,c1,driftinv)
```

Here, `driftinv` counts the number of periods needed for a clock to exceed the other one by one unit.

4.3 The whole system

The whole system is shown at table 5. It is made of the two trains and their respective controllers. Each controller has its own clock, the environment (the two trains) also has a clock, and these clocks are assumed to share the same period.

4.4 Communication abstraction

However, this system, when translated to PVS, is too large to be proved. A first step toward a proof consists of abstracting communication. For this we consider a variable `v`, computed on clock `c`, communicated to another subsystem working on clock `c'` and then re-sampled on clock `c`:

```
v' = (current(c',v) ((current(c,v) v) when c')) when c
```

Then we prove the property:

Theorem 1 (Sampling) `sampling : THEOREM`

```
  (FORALL (n:nat): same_period(c,c',1)(n))
   => FORALL (n:nat):
     ((v' = v) or (v'= fby(v,v)) or (v'= fby(v,fby(v,v))))(n)
```

meaning that the value used by another process cannot be two time unit older than the value used by the process issuing it.

Now, this property has an interesting corollary, when applied to smooth signals, i.e. signals with bounded variations. This corollary is as follows:

Theorem 2 (Smooth sampling) `smooth_sampling : THEOREM`

```
  (FORALL (n:nat): abst_val(v - fby(v, v))(n) < e )
     => FORALL (n:nat):  abst_val(v'- v)(n) < 2*e
```

4.5 The system revisited

When trying to apply these results to the system, we meet two obstacles:

- train speeds are not upper-bounded and, thus, train positions are not smooth. We must redesign the controller so as to enforce speed limits:

  ```
  const vlim : real;
  assert vlim > 0;
  fu  = df((x2  fby (d - v*dt)) - amax*dt*dt,
           (0 fby v) + amax*dt
          )< 0
         or (0 fby v) + amax*dt > vlim;
  ```

- even in case of speed limits, we still cannot prove the system; this is due to the fact that the difference between the train position computed by the controller and the one "computed" by the train cannot be upper-bounded. Position computation is an integral and can be seen as a non-stable system[2]. Such a system sums up input and integration errors and two such systems (the "real" and the computed one) put in parallel, will sooner or later diverge. This is why train controllers need, from time to time to be given the actual position, for instance thanks to beacons located on the track. However, taking into account these features is complicated and goes far away from the object of the paper. We shall thus limit ourselves to assuming that each train controller always samples its position.

We can now compute :

[2] In fact a limit-stable system.

Speed errors: Defining

```
v1c = (current (c1,0) (v1 when c1)) when ct;
```

i.e. the image in the "train" of the speed as sampled by the controller, we can prove:

```
speed_error: LEMMA FORALL (n:nat):
  v1(n) <= v1c(n) + 2*amax*dt
```

Distance errors: Similarly, letting

```
x1c = (current (c1,0) (x1 when c1)) when ct;
```

be the image, in the "train", of the position as sampled by the controller, we can prove:

```
distance_error: LEMMA FORALL (n:nat):
  x1(n) <= x1c(n) + 2*vlim*dt
  AND
  x1(n) >= x1c(n)
```

Finally, the control program which allows us to prove the correctness property is:

```
node control (am,v,x2,x1:real)
     returns (a,xp:real);
var d:real; fu: bool;
let
  d   = x2 - x1 -2*vlim*dt;
  fu  = df((x2  fby (d - v*dt)) - 3*amax*dt*dt,
           (0 fby v) + 3*amax*dt
          )< 0
        or (0 fby v) + 3*amax*dt > vlim ;
  a   = if fu then -afu else am ;
  xp = x1 ;
tel
```

5 Conclusion

In this paper, we intended to show how to use the synchronous data-flow language Lustre, combined with the PVS proof system in deriving a proved (distributed) control program. We hopefully illustrated the features of our approach :

- asynchronous periodic programs with nearly the same period, communicating by sampling and well taken into account by Lustre,
- equational reasoning which leaves to the Lustre compiler the task of scheduling computations,

- no distinction between control programs and physical environments which are sampled in the same way. This allows us to provide "elementary" proofs based on difference equations instead of differential ones which require more involved PVS formalization.

However the approach is only partially automated: the translation tool from Lustre to PVS does not yet handle sampling and holding primitives. A new, more complete and fully functional version based on Kahn's theory [13] is in progress.

Furthermore, it can be objected that this sampling-based approach may be better adapted to numerical problem, like the one presented here, than to logical ones involving complex state machines. This will be a subject for future work.

Finally the question of whether this experiment can be extended to larger applications and other domains is left open. Let us simply note here that the chosen example is a favorable one, in the sense that it bears simple, logical specifications. This may not be the case of most control systems. For instance a specification for an aircraft controller could be "keep the aircraft within its flight domain", and this is surely a less favorable case.

Acknowledgment The authors acknowledge Daniel Pilaud from INRIA who initiated this work, and Oder Maler from Verimag for friendly discussions on hybrid systems.

References

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1995.

[2] J.L. Bergerand and E. Pilaud. SAGA; a software development environment for dependability in automatic control. In *SAFECOMP'88*. Pergamon Press, 1988.

[3] R.S. Boyer and J.S.Moore. *A Computational Logic*. Academic Press, 1979.

[4] D. Brière, D. Ribot, D. Pilaud, and J.L. Camus. Methods and specification tools for Airbus on-board systems. In *Avionics Conference and Exhibition*, London, December 1994. ERA Technology.

[5] P. Caspi and N. Halbwachs. A functional model for describing and reasoning about time behaviour of computing systems. *Acta Informatica*, 22:595–627, 1986.

[6] T. Coquand and G. Huet. The calculus of construction. *Information and Computation*, 76(2), 1988.

[7] E. Doginova and N. Lynch. Safety verification fo automated platoon maneuvers: A case study. In O. Maler, editor, *Hybrid and real-Time Systems, Grenoble*, volume 1201 of *Lecture Notes in Computer Science*, pages 154–170. Springer, 1997.

[8] M. J. Gordon and T. F. Melham. *Introduction to HOL : A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[10] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous dataflow language LUSTRE. *IEEE Trans. on Software Engineering*, 18(9):785–793, September 1992. Available through anonymous ftp at `imag.fr:pub/SPECTRE/LUSTRE/PAPERS/lustre.tse.ps.gz`.

[11] U. Hensel and B. Jacobs. Coalgebraic theories of sequences in pvs. Technical Report CSI-R9708, Computer Science Institute, University of Nijmegen, 1997. available at `http://www.cs.kun.nl/ bart/PAPERS/`.

[12] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, 1990.

[13] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74*. North Holland, 1974.

[14] K.J.Åström and B.Wittenmark. *Computer Controlled Systems*. Prentice-Hall, 1984.

[15] L.Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. and Sys.*, 16(3), 1994.

[16] P.S. Miner and S.D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit. In Mary Sheeran and Satnam Singh, editors, *Designing Correct Circuits*, Electronic Workshops in Computing, Bastad, Sweden, September 1996. Springer-Verlag.

[17] N.Shankar, S.Owre, and J.Rushby. The PVS Proof Checker : A Reference Manual. march 1993.

[18] S. Owre, J. Rushby, and N. Shankar. PVS: a prototype verification system. In *11th Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Verlag, 1992.

[19] C. Paulin-Mohring. Circuits as streams in Coq, verification of a sequential multiplier. Research Report 95-16, Laboratoire de l'Informatique du Parallélisme, September 1995. Available at `http://www.ens-lyon.fr:80/LIP/lip/publis/`.