A PVS Proof Obligation Generator for LUSTRE Programs

(Unpublished Draft)

Cécile Dumas
Verimag

November 1999

1 Introduction

Recently, several tool-boxes for program formal derivation and proof have been proposed, among which we can cite B [1], COQ [3], PVS [10] and HOL [4]. Clearly this growing interest is due to the also growing involvement of software in critical systems, and the consequent concern about software errors.

In these domains, synchronous data-flow programming (such as the LUSTRE language [5]) has become a popular approach. This has entailed a need for formal methods adapted to this language. A long lasting effort has already been made in this sense, mostly based on "model checking" techniques [6]. Model-checking was a good choice since automatic control programs (and more generally reactive programs) exhibit a control structure made of a finite automaton, over which most properties are decidable.

Yet there are many control problems which don't fall into such class, and should be handled within more general deductive frameworks like the ones cited above.

The object of the paper is to present a tool intended to help in applying the PVS approach to LUSTRE programs. The method must be efficient, flexible (for incomplete program and modular proofs) and user-friendly. We adopt a solution based on Kahn's semantic [7] and on mu-calculus [8] as properties language. First we shall explain how we use this two theories. Then, the developed tool prototype will be presented.

2 Kahn Theory of Data-Flow

Flows are represented by finite or infinite sequences, built as follow :

- ϵ is the empty sequence.

- If v is an element of type T and s a finite or infinite sequence of elements of type T ($s = (s_0, \ldots, s_n, \ldots)$), then $v \cdot s$ represent the sequence $(v, s_0, \ldots, s_n, \ldots)$.

Moreover, the first element of a non empty sequence ss is $hd(ss)$ and the remainder is $tl(ss)$. The set of these finite or infinite sequences is called D^∞.

We define a partial order on D^∞ :

$$\forall x, y \in D^\infty, (x \leq y) \Leftrightarrow (\exists z, y = x \bullet z)$$

where $x \bullet z$ represent the concatenation of sequences x et z ($\leq$ is the prefix order over sequences). We then have the following results:

- $(D^\infty, \leq, \epsilon)$ is a Complete Partial Order (CPO) and a set of vectors of D^∞ is a CPO.
- **Kleene's theorem** : In a CPO, continuous function have a least fixed point.

We can now prove that LUSTRE functions and nodes are continuous. So the behavior of a LUSTRE node corresponds to a least solution of the set of equations, in the sense of the prefix order over sequences. For example:

```
node count(reset:bool;(val:int)when reset) returns (res:int);
let res = if reset
          then current val
          else pre res - 1;
tel;
```

is represented by the function :

$$(reset, val) \longmapsto \mu[res \longmapsto \text{if } reset \text{ then current } val \text{ else pre } res - 1]$$

and

```
node mux(m:int) returns (c:bool;y:int);
var x:int;
let assert( m >= 0 );
    y     =       count(c,x);
    c     =       true -> (pre y = 0);
    x     = m when c;
tel;
```

is represented by the function :

$$m \longmapsto \mu[(c, y, x) \longmapsto (\text{true } \rightarrow (\text{pre } y = 0), \text{count}(c, x), m \text{ when } c)]$$

This representation has two interests :

- equations don't need to be sorted for flows calculus, (for example, the order of flows calculus of `mux` is `c`, then `x`, then `y`)
- LUSTRE programs don't need to be compiled, because deadlocks have solutions: ϵ.

3 Linear Mu-calculus

We consider fixed point equations on the complete lattice of predicates:

$$(\{f,t\}^{D^\infty}, \subseteq, \emptyset, D^\infty)$$

Monotonic functions have unique least (`Mu`) and greatest (`Nu`) fixed points, and we have the safety property model checking :

For any continuous predicates Q and for any increasing chain $C = s_1 \leq \ldots s_n \leq \ldots$

$$Q(supC) = \cap_{s_n \in C} Q(s_n)$$

Then the proof rule used to prove LUSTRE properties is :

For any continuous predicates Q, and continuous functions f :

$$\frac{\begin{array}{l} Q(\epsilon) \\ \forall x, Q(x) \Rightarrow Q(f(x)) \end{array}}{Q(\mu f)}$$

4 The Proof Obligation Generator

Initially, we defined Kahn's semantic and Mu-calculus in PVS, but trying to directly use these theories through PVS strategies was rather inefficient. So we decided to apply the strategies off-line and to only leave to PVS the remaining proof obligations.

We use the following rules which hold for any continuous predicates F :

merge rule :

$$\frac{\mathcal{H}}{F(\texttt{current}(v,x,c))} \Leftarrow \frac{\begin{array}{c} \mathcal{H} \\ F(y) \end{array}}{F(\texttt{mer}(x, v \cdot y, c))}$$

where

$$\texttt{mer}(x,y,c) = \texttt{if hd } c \quad \begin{array}{l} \texttt{then hd } x \cdot \texttt{mer}(\texttt{tl } x, \texttt{tl } y, \texttt{tl } c) \\ \texttt{else hd } y \cdot \texttt{mer}(x, \texttt{tl } y, \texttt{tl } c) \end{array}$$

derive rule :

$$\frac{\mathcal{H}}{F(\texttt{let rec } x_1 = e_1, \ldots, x_n = e_n \texttt{in } e)} \Leftarrow \frac{\begin{array}{c}\mathcal{H}\\ F(e)\end{array}}{F(\texttt{let rec } x_1 = e_1, \ldots, x_n = e_n \texttt{in } e[e_i/x_i])}$$

develop rule :

$$\frac{\mathcal{H}}{\texttt{All}(f)} \Leftarrow \underbrace{\frac{\mathcal{H}}{\texttt{hd } f}}_{\text{proof obligation for PVS}} \wedge \frac{\mathcal{H}}{\texttt{All}(\texttt{tl } f)}$$

where

$$\begin{array}{rccc}\texttt{All}: & \mathit{bool}^{\infty} & \rightarrow & \mathit{bool} \\ & s & \longmapsto & \texttt{hd } s \wedge \mathit{All}(\texttt{tl } s)\end{array}$$

compare rule :

$$\frac{\mathcal{H}}{F(x)} \Leftarrow \frac{\mathcal{H}'}{F(\texttt{tl } x)}$$

These rules are combined in order to get an optimal strategy as shown in figure 1.

5 Examples

Example 1: Consider the LUSTRE program:

```
node compteur(x:int) returns (y:int);
let assert( x >= 0 );
    y     =       0 -> pre y + x;
tel;
```

We want to prove :

$$\frac{x \geq 0}{\texttt{let rec } y = 0 \cdot y + x \texttt{in } y \geq 0}$$

It is not possible to apply the merge rule, but we can apply the derive rule :

$$\frac{\begin{array}{c} y \geq 0 \\ x \geq 0 \end{array}}{\text{let rec } y = 0 \cdot y + x \text{ in } 0 \cdot y + x \geq 0}$$

As y is derived from y, we stop the derivation, in order not to loop. There are no clocks in the consequent, so we can explicit the head of this sequence :
The first sub-consequent :

$$\frac{\begin{array}{c} y \geq 0 \\ x \geq 0 \end{array}}{0 + \text{hd } x \geq 0}$$

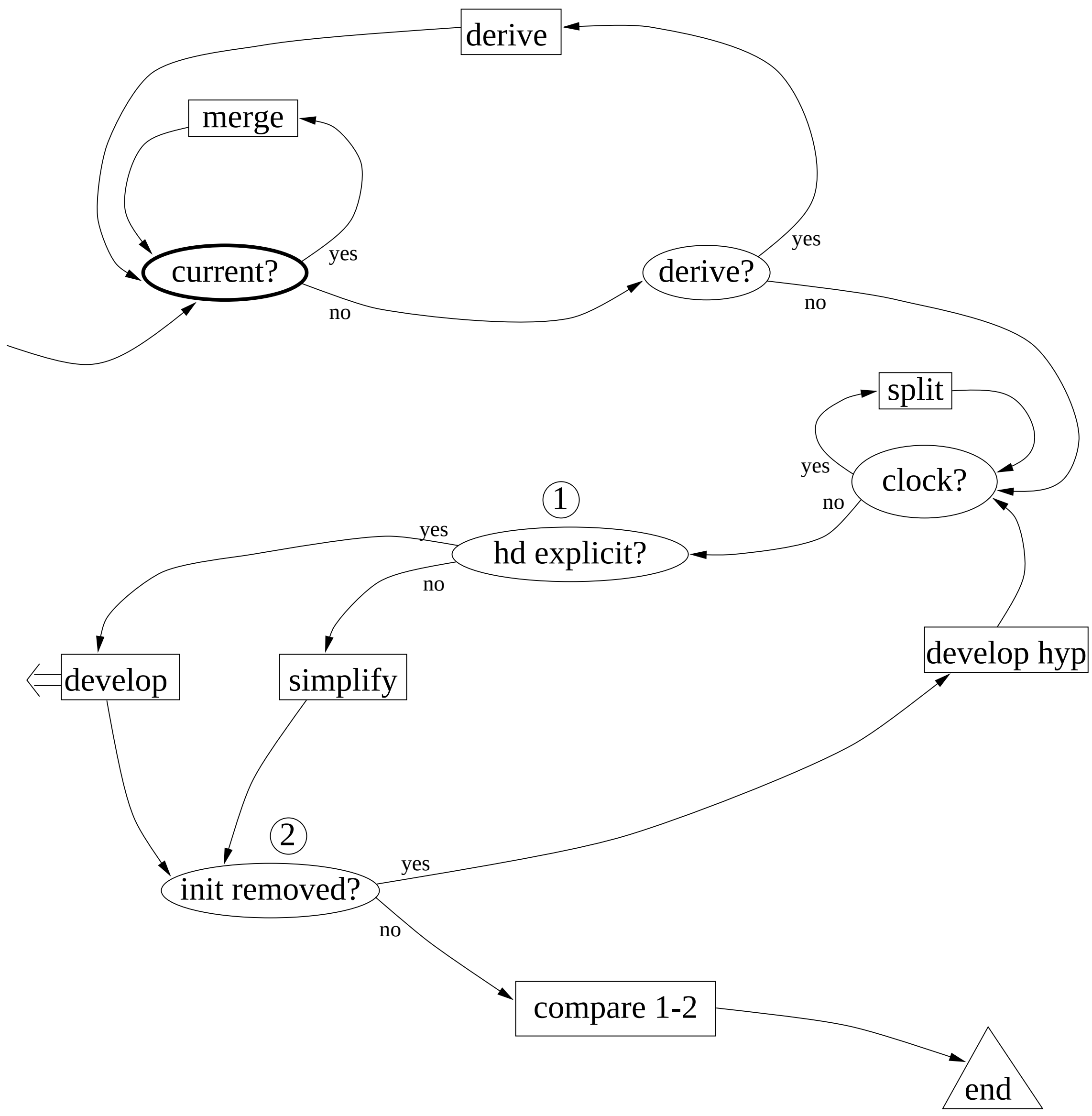

Figure 1: Rule scheduling

is simply proved, because $x \geq 0 \Rightarrow \text{hd } x \geq 0$. In the second sub-consequent :

$$\frac{\begin{array}{c} y \geq 0 \\ x \geq 0 \end{array}}{y + \text{tl } x \geq 0}$$

the initialization $0 \cdot y$ has been removed, so we develop the hypothesis.

$$\frac{\begin{array}{c} y \geq 0 \\ x \geq 0 \\ \text{tl } y \geq 0 \\ \text{tl } x \geq 0 \end{array}}{y + \text{tl } x \geq 0 \quad (1)}$$

Then we develop the consequent in two sub-consequents :
The first sub-consequent :

$$\frac{\begin{array}{c} y \geq 0 \\ x \geq 0 \\ \text{tl } y \geq 0 \\ \text{tl } x \geq 0 \end{array}}{\text{hd } y + \text{hd tl } x \geq 0}$$

is simply proved, because $\text{tl } x \geq 0 \Rightarrow \text{hd tl } x \geq 0$ and $y \geq 0 \Rightarrow \text{hd } y \geq 0$. In the second sub-consequent :

$$\frac{\begin{array}{c} y \geq 0 \\ x \geq 0 \\ \text{tl } y \geq 0 \\ \text{tl } x \geq 0 \end{array}}{\text{tl } y + \text{tl tl } x \geq 0 \quad (2)}$$

no initialization is removed, so we compared expressions (1) and (2), that have an identical gap on each variable (here one for x and one for y).

Here is the PVS file obtained with the tool :

```
compteur_node : THEORY

  BEGIN

  IMPORTING top_lustre,const

  x : VAR suite[int] y : VAR suite[int]

  compteur_prop1_0 :
    THEOREM (   (hd(y)  >= 0) AND (hd(x)  >= 0) )
               => (0 >= 0)
```

```
  compteur_prop1_00 :
    THEOREM (   (hd(tl(y))  >= 0) AND (hd(tl(x))  >= 0)
               AND (hd(y)  >= 0) AND (hd(x)  >= 0) )
                 => ((hd(y)  + hd(tl(x)) ) >= 0)

END compteur_node
```

Example 2: Let the LUSTRE program:

```
node filtre(c:bool,(x when c):int) returns (y:int);
var z: int;
let assert( x >= 0 );
    y     =      current(0,x,c);
tel;
```

We want to prove :

$$\frac{x \geq 0}{\text{let rec } y = \text{current}(0, x, c) \text{ in } y \geq 0}$$

It is not possible to apply the merge rule, but we can apply the derive rule :

$$\frac{\begin{array}{c} y \geq 0 \\ x \geq 0 \end{array}}{\text{let rec } y = \text{current}(0, x, c) \text{ in current}(0, x, c) \geq 0}$$

We can now apply the merge rule :

$$\frac{\begin{array}{c} cur \geq 0 \\ y \geq 0 \\ x \geq 0 \end{array}}{\text{mer}(x, 0 \cdot cur, c) \geq 0}$$

The consequent has a clock hd c, we obtain two sub-consequents :

$$\frac{\begin{array}{c} \text{hd } c \\ cur \geq 0 \\ y \geq 0 \\ x \geq 0 \end{array}}{\text{mer}(x, 0 \cdot cur, c) \geq 0} \quad \text{et} \quad \frac{\begin{array}{c} \overline{\text{hd } c} \\ cur \geq 0 \\ y \geq 0 \\ x \geq 0 \end{array}}{\text{mer}(x, 0 \cdot cur, c) \geq 0}$$

These consequents contain no clock and can be develop each one in two sub-consequents :

Head and tail of the first consequent :

$$\underbrace{\frac{\begin{array}{c}\text{hd } c\\ cur \geq 0\\ y \geq 0\\ x \geq 0\end{array}}{\text{hd } x \geq 0}}_{\text{this is simply proved}} \quad \text{and} \quad \frac{\begin{array}{c}\text{hd } c\\ cur \geq 0\\ y \geq 0\\ x \geq 0\end{array}}{\text{mer}(\text{tl } x, cur, \text{tl } c) \geq 0}$$

and head and tail of the second one :

$$\underbrace{\frac{\begin{array}{c}\overline{\text{hd } c}\\ cur \geq 0\\ y \geq 0\\ x \geq 0\end{array}}{0 \geq 0}}_{\text{which is trivial...}} \quad \text{and} \quad \frac{\begin{array}{c}\overline{\text{hd } c}\\ cur \geq 0\\ y \geq 0\\ x \geq 0\end{array}}{\text{mer}(x, cur, \text{tl } c) \geq 0}$$

In remaining consequents, the initialization $0 \cdot cur$ are removed, so we develop the hypothesis.

$$\frac{\begin{array}{c}\text{hd } c\\ cur \geq 0\\ y \geq 0\\ x \geq 0\\ \text{tl } cur \geq 0\\ \text{tl } y \geq 0\\ \text{tl } x \geq 0\end{array}}{\text{mer}(\text{tl } x, cur, \text{tl } c) \geq 0 \ \ (3)} \quad \text{and} \quad \frac{\begin{array}{c}\overline{\text{hd } c}\\ cur \geq 0\\ y \geq 0\\ x \geq 0\\ \text{tl } cur \geq 0\\ \text{tl } y \geq 0\\ \text{tl } x \geq 0\end{array}}{\text{mer}(x, cur, \text{tl } c) \geq 0 \ \ (4)}$$

A new clock hd tl c appears, that give four sequents. Then we develop each consequent in two consequents :

Case hd c and hd tl c :

$$\underbrace{\frac{\begin{array}{c}\text{hd tl } c\\ \text{hd } c\\ cur \geq 0\\ y \geq 0\\ x \geq 0\\ \text{tl } cur \geq 0\\ \text{tl } y \geq 0\\ \text{tl } x \geq 0\end{array}}{\text{hd tl } x \geq 0}}_{\text{that is true}} \quad \text{and} \quad \underbrace{\frac{\begin{array}{c}\text{hd tl } c\\ \text{hd } c\\ cur \geq 0\\ y \geq 0\\ x \geq 0\\ \text{tl } cur \geq 0\\ \text{tl } y \geq 0\\ \text{tl } x \geq 0\end{array}}{\text{mer}(\text{tl tl } x, \text{tl } cur, \text{tl tl } c) \geq 0}}_{\text{which is synchron with (3)}}$$

Case hd c and $\overline{\text{hd tl } c}$:

$$\underbrace{\frac{\begin{array}{c}\overline{\text{hd tl } c}\\ \text{hd } c\\ cur \geq 0\\ y \geq 0\\ x \geq 0\\ \text{tl } cur \geq 0\\ \text{tl } y \geq 0\\ \text{tl } x \geq 0\end{array}}{\text{hd } cur \geq 0}}_{\text{that is true}} \quad \text{and} \quad \underbrace{\frac{\begin{array}{c}\overline{\text{hd tl } c}\\ \text{hd } c\\ cur \geq 0\\ y \geq 0\\ x \geq 0\\ \text{tl } cur \geq 0\\ \text{tl } y \geq 0\\ \text{tl } x \geq 0\end{array}}{\text{mer}(\text{tl } x, \text{tl } cur, \text{tl tl } c) \geq 0}}_{\text{which is synchron with (3)}}$$

Case $\overline{\text{hd } c}$ and hd tl c :

$$\underbrace{\frac{\begin{array}{c}\text{hd tl } c\\ \overline{\text{hd } c}\\ cur \geq 0\\ y \geq 0\\ x \geq 0\\ \text{tl } cur \geq 0\\ \text{tl } y \geq 0\\ \text{tl } x \geq 0\end{array}}{\text{hd } x \geq 0}}_{\text{that is true}} \quad \text{and} \quad \underbrace{\frac{\begin{array}{c}\text{hd tl } c\\ \overline{\text{hd } c}\\ cur \geq 0\\ y \geq 0\\ x \geq 0\\ \text{tl } cur \geq 0\\ \text{tl } y \geq 0\\ \text{tl } x \geq 0\end{array}}{\text{mer}(\text{tl } x, \text{tl } cur, \text{tl tl } c) \geq 0}}_{\text{which is synchron with (4)}}$$

Case $\overline{\text{hd } c}$ and $\overline{\text{hd tl } c}$:

$$\underbrace{\frac{\begin{array}{c}\overline{\text{hd tl } c}\\ \overline{\text{hd } c}\\ cur \geq 0\\ y \geq 0\\ x \geq 0\\ \text{tl } cur \geq 0\\ \text{tl } y \geq 0\\ \text{tl } x \geq 0\end{array}}{\text{hd } cur \geq 0}}_{\text{that is true}} \quad \text{and} \quad \underbrace{\frac{\begin{array}{c}\overline{\text{hd tl } c}\\ \overline{\text{hd } c}\\ cur \geq 0\\ y \geq 0\\ x \geq 0\\ \text{tl } cur \geq 0\\ \text{tl } y \geq 0\\ \text{tl } x \geq 0\end{array}}{\text{mer}(x, \text{tl } cur, \text{tl tl } c) \geq 0}}_{\text{which is synchron with (4)}}$$

Here is the PVS theory automatically generated with the tool :

```
filtre_node : THEORY

  BEGIN
```

```
IMPORTING top_lustre,const

x : VAR suite[int] c : VAR suite[bool] y : VAR suite[int]

filtre_prop1_1 : THEOREM
  FORALL (mem?1 : suite[int]) : (   (hd(c) )
    AND (hd(mem?1)  >= 0)
      AND (hd(y)  >= 0)
        AND (hd(x)  >= 0) )
          => (hd(x)  >= 0)

filtre_prop1_11 : THEOREM
  FORALL (mem?1 : suite[int]) : (   (hd(tl(c)) )
    AND (hd(c) )
      AND (hd(tl(mem?1))  >= 0)
        AND (hd(tl(y))  >= 0)
          AND (hd(tl(x))  >= 0)
            AND (hd(mem?1)  >= 0)
              AND (hd(y)  >= 0)
                AND (hd(x) >= 0) )
                  => (hd(tl(x))  >= 0)

filtre_prop1_12 : THEOREM
  FORALL (mem?1 : suite[int]) : ( NOT(hd(tl(c)) )
    AND (hd(c) )
      AND (hd(tl(mem?1))  >= 0)
        AND (hd(tl(y)) >= 0)
          AND (hd(tl(x))  >= 0)
            AND (hd(mem?1)  >= 0)
              AND (hd(y)  >= 0)
                AND (hd(x)  >= 0) )
                  => (hd(mem?1)  >= 0)

filtre_prop1_2 : THEOREM
  FORALL (mem?1 : suite[int]) : (   NOT(hd(c) )
    AND (hd(mem?1)  >= 0)
      AND (hd(y)  >= 0)
        AND (hd(x)  >= 0) )
          => (0 >= 0)

filtre_prop1_21 : THEOREM
  FORALL (mem?1 : suite[int]) : (   (hd(tl(c)) )
    AND NOT(hd(c) )
      AND (hd(tl(mem?1))  >= 0)
        AND (hd(tl(y))  >= 0)
          AND (hd(tl(x))  >= 0)
```

```
            AND (hd(mem?1)  >= 0)
              AND (hd(y)  >= 0)
                AND (hd(x) >= 0) )
                  => (hd(x)  >= 0)

  filtre_prop1_22 : THEOREM
    FORALL (mem?1 : suite[int]) : ( NOT(hd(tl(c)) )
      AND NOT(hd(c) )
        AND (hd(tl(mem?1))  >= 0)
          AND (hd(tl(y))  >= 0)
            AND (hd(tl(x))  >= 0)
              AND (hd(mem?1)  >= 0)
                AND (hd(y)  >= 0)
                  AND (hd(x)  >= 0) )
                    => (hd(mem?1) >= 0)


END filtre_node
```

6 Conclusion

The problem of dealing with clocks leads to consider either "absent" elements and a relational semantic [2, 9] or both finite and infinite sequences. As we aimed at staying as near as possible from the source language, we decided to choose the second solution and, fortunately, Kahn semantic combined with linear mu-calculus allowed us to achieve this goal.

Yet, defining PVS strategies to automatically handle the LUSTRE to PVS translation appeared very inefficient. This is why we chose to run these strategies off-line, in the translation tool itself (written in Caml), and to only generate PVS proof obligations, that is to say, what remains to prove, once the unfolding of inference rules has been performed.

Then, this tool appears very efficient: it allows us to handle multiple clocks in a functional style, it gives us the same proof obligations that resulted from the old translation tool, but much faster and it sometimes perform even better, in the sense that it often finds the wright number of unfoldings which allows a property to be proved.

Furthermore, it offers the possibility of dealing with asynchronous and possibly unbounded Kahn networks, and thus the possibility to model and prove mixed synchronous-asynchronous designs. But this has still to be developed.

Several other points remain to be studied:

- Experiments have only dealt with safety properties. Liveness ones have to be addressed.

- Only flat programs have been studied. This is likely to only apply to small programs and raises the question of addressing large, structured ones.
- One way of answering this last question could be to study some refinement-based design method *à la* B, adapted to the synchronous programming context. In this sense, a proof obligation generator like the one described here could be a good starting point.

References

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

[2] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. In *Proceedings of the Workshop on Coalgebraic Methods in Computer Science, Lisbon*, volume 11 of *Electronic Notes in Theoretical Computer Science*. Elsevier, March 1998.

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

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

[5] 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.

[6] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous dataflow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, September 1992.

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

[8] D. Kozen. Results on the propositional mu-calculus. *Theor.Comp.Sci*, 27:333–354, 1983.

[9] D. Nowak, J.-R. Beauvais, J.-P. Talpin. Co-Inductive Axiomatization of a Synchronous Language. In *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics*, number 1479 in LNCS, pages 387–399 Springer Verlag, October 1998.

[10] 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.