A PVS Proof Obligation Generator for Lustre Programs*

Cécile Canovas-Dumas and Paul Caspi
{cdumas,caspi}@imag.fr

Laboratoire Verimag (CNRS, UJF, INPG)
http://www-verimag.imag.fr

Abstract. This paper presents a tool for proving safety properties of Lustre programs in PVS, based on continuous induction. The tool applies off-line a repeated induction strategy and generates proof obligations left to PVS. We show on examples how it avoids some drawbacks of co-induction which needs to consider "absent elements" in the case of clocked streams.

1 Introduction

Co-induction has been advocated as providing a good theoretical framework for proving stream programs and several experiments and tools [5, 17, 13, 9, 14, 7, 2] have been recently designed in this setting, mostly based on Coq [6] and PVS [15]. Two main proof principles have been used, the "Bisimulation Proof Principle" originated from Park's work and the "Infinite Proof Principle" due to Coquand. However, when we tried to apply these principles to the proof of Lustre [8] programs, we found that they were less efficient than the one which arises from the old, semantic based, Kahn's theory of data-flow[11].

This observation mainly arises when considering multi-clock systems where sub-streams are extracted from streams by some filtering procedure. A way of dealing with such sub-streams within the co-inductive framework consists of introducing an "absent" element which takes the place of erased elements in the filtering process.[1] The point is that, as we show in section 5, bisimulation proofs rely on the observations (destructors of the co-inductive type) we can draw from a running process, and an "absent" element is sometimes a quite poor observation of what is actually taking place in the process. This may oblige us to add the observation of some internal state values or, equivalently, to strengthen the property we want to prove by extending it to these state values. As for infinite proofs, "absent" elements require more complex case analysis.

An alternate solution arises from the remark that stream programming, as found in Lustre, describes programs as functions from streams to streams. The

* This work has been partially supported by Esprit project Syrf and Inria action Presysa.

[1] This absent element is the analog of "silent" elements introduced by Milner in synchronizing trees [12].

co-inductive point of view comes from the fact that the co-domain type of our functions (streams) is a co-inductive type. Now another possible proof principle can be derived from the recursive nature of the domain type (also streams). Induction allows us to prove properties of functions over inductive types, but these, in general only yield finite objects. This is where continuity is helpful, by bridging the gap between such finite objects and the infinite ones (streams) we deal with. When taking this point of view, we shall see that we don't anymore need "absent" elements. Thus every observation we draw from a process is meaningful and proofs get simpler and shorter.

In order to exemplify this observation, we shall first say some words of stream programming based on Lustre (section 2). Section 3 shows how to prove safety properties of Lustre programs by continuous induction and section 4 briefly describes the tool we designed for automatically generating PVS proof obligations from Lustre programs and safety properties. In section 5 we provide a tentative comparison with co-induction and finally discuss related works.

2 Lustre and Stream Programming

2.1 Kahn Semantic

The main idea of Kahn network semantic is to consider :

- The complete partial order (CPO) $(D^\infty, \leq, \epsilon)$ where $D^\infty = D^* + D^\omega$ is the set of finite and infinite sequences of some set D, with respect to the prefix order of sequences $x \leq y \Rightarrow \exists z : y = x@z$ and the empty sequence $x = \epsilon@x = x@\epsilon$, where @ is the concatenation of sequences.
  Completeness, here, amounts to the fact that every chain $C = \{x_0 \leq x_1 \ldots \leq x_n \ldots\}$ has a least upper bound $lub\ C$.
- The CPO of functions over D^∞ tuples and higher order extensions.
- Least fixed points of continuous functions, where continuity refers to lub preservation. In case of higher order functions, these fixed points are known to be continuous functions.

In this setting, continuity implies monotony which, in turn, can be interpreted as causality: if x is a prefix of y, then $(f\ x)$ is a prefix of $(f\ y)$.

2.2 Lustre Primitives

Table 1 displays the inductive definitions of Lustre primitives. In this table:

- . is the usual sequence constructor;
- any constant is lifted to infinite sequences and similarly, every operator is lifted to operate point-wise on sequences;
- `->pre` is a unit delay;
- `when` is a filtering operator;
- `current` is a converse hold operator.

It is easy to show that every primitive is continuous and every composition of primitives is continuous.

xs `+` ϵ	$= \epsilon$
ϵ `+` ys	$= \epsilon$
$x.xs$ `+` $y.ys$	$= (x+y).(xs$ `+` $ys)$
ϵ `->pre` ys	$= \epsilon$
$x.xs$ `->pre` ϵ	$= x.\epsilon$
$x.xs$ `->pre` $y.ys$	$= x.(y.ys$ `->pre` $ys)$
ϵ `when` cs	$= \epsilon$
xs `when` ϵ	$= \epsilon$
$x.xs$ `when` $true.cs$	$= x.(xs$ `when` $cs)$
$x.xs$ `when` $false.cs$	$= xs$ `when` cs
`current` $v\ \epsilon\ xs$	$= \epsilon$
`current` $v\ false.cs\ xs$	$= v.($`current` $v\ cs\ xs)$
`current` $v\ true.cs\ \epsilon$	$= \epsilon$
`current` $v\ true.cs\ x.xs$	$= x.($`current` $x\ cs\ xs)$

Table 1. Inductive definition of Lustre primitives

2.3 Lustre Programs

Then Lustre programs are sets of (mutually recursive) definitions built on these primitives. For instance, `fib` defines the sequence of Fibonacci numbers:

```
fib = 1 -> pre(fib +(0 -> pre fib));
```

Functions (nodes) allow definitions to be encapsulated:

```
node sum(delta, x : real) returns (y : real);
        let
             y = x*delta + (0 -> pre y);
        tel
```

in such a way that `z = sum(2.0*delta, x when half)` represents an integrator operating half rate. This functional style makes Lustre a useful language for programming control and hardware systems.

2.4 Synchrony

Though not essential here, let us say a word on synchrony: it consists of rejecting, thank to some statics analysis referred to as "clock calculus", expressions like `x + (x when half)` whose execution requires an unbounded memory.

3 Induction and Continuity

The idea is to look at safety properties (*All* modality) as inductively defined predicates over input sequences. Yet, infinite sequences are not an algebraic type and cannot be considered as an initial algebra but, as we already noticed, they can be considered as *lub*s of finite sequence chains. If our predicates are continuous and if we prove that a property holds for every finite input sequence, it will hold by continuity on the *lub*s.

Table 2 displays the proof rules for such safety properties:

- the rules assume x, xs and xss are not free in H;
- (IND) is the ordinary induction on lists;
- (REC) is the usual fix-point rule, which also extends by continuity.

$$(EMPTY)\frac{}{H \vdash All\ \epsilon} \qquad (CONC)\frac{H \vdash x \quad H \vdash All\ xs}{H \vdash All\ x.xs}$$

$$(HYP1)\frac{}{H, All\ x.xs \vdash x} \qquad (HYP2)\frac{}{H, All\ x.xs \vdash All\ xs}$$

$$(IND)\frac{\begin{array}{c} H \vdash (h\ \epsilon) \\ H, (h\ xs) \vdash (h\ x.xs) \end{array}}{H \vdash (h\ xss)} \qquad (REC)\frac{\begin{array}{c} H \vdash (h\ \epsilon) \\ H, (h\ x) \vdash (h\ (e\ x)) \end{array}}{H \vdash h\ (rec\ x : (e\ x))}$$

Table 2. Proof rules

Example 1: We want to prove: $\forall xs, cs : All\ xs \Rightarrow All\ (\texttt{current}\ true\ cs\ xs)$

First step: we can choose here to induct on cs. This gives three sub-goals:

$$\frac{All\ xs}{All\ (\texttt{current}\ true\ \epsilon\ xs)},$$ which holds by expanding `current`, and

$$\frac{\begin{array}{l} All\ xs \\ \forall xs : All\ xs \Rightarrow All\ (\texttt{current}\ true\ cs\ xs) \end{array}}{All\ (\texttt{current}\ true\ false.cs\ xs)},$$ which holds by expanding `current` and using the induction hypothesis, and

$$\frac{\begin{array}{l} All\ xs \\ \forall xs : All\ xs \Rightarrow All\ (\texttt{current}\ true\ cs\ xs) \end{array}}{All\ (\texttt{current}\ true\ true.cs\ xs)},$$ which yields, by expanding `current` and remarking that the $xs = \epsilon$ case holds directly:

$$\frac{\begin{array}{l} All\ x.xs \\ \forall xs : All\ xs \Rightarrow All\ (\texttt{current}\ true\ cs\ xs) \end{array}}{All\ (\texttt{current}\ x\ cs\ xs)}$$

General step: at this point, we have got rid of initialization and deal with the general case which can be restated as
$\forall x, xs : All\ x.xs \Rightarrow All\ (\texttt{current}\ x\ cs\ xs)$

By induction on cs, we get three new goals; the ϵ and $false.cs$ work as previously, and the last one leads to:

$\dfrac{\begin{array}{l} All\ x.xs \\ \forall x, xs : All\ x.xs \Rightarrow All\ (\texttt{current}\ x\ cs\ xs) \end{array}}{All\ (\texttt{current}\ x\ true.cs\ xs)}$. The $xs = \epsilon$ case holds directly and the $xs = x'.xs'$ case holds by instantiating the induction hypothesis.

We can see here that we have exactly examined four cases at each step, the initial step and the general case step and this is clearly the minimum number of cases needed to prove the property. Furthermore, the cases involving ϵ are obvious and yield direct proofs.

4 A Proof Obligation Generator for Lustre Programs

Initially, we defined Kahn semantic in PVS[15], but trying to directly use this theory through PVS strategies is rather inefficient.[2] So we decided to unfold the strategies off-line and to only leave to PVS the remaining proof obligations.

For instance a proof goal like $\dfrac{H}{All\ x.xs}$ will generate the proof obligation $\dfrac{H}{x}$ left to PVS, while the remaining goal $\dfrac{H}{All\ xs}$ will be analyzed off-line. Furthermore ϵ-like goals will be in general discharged within the tool and do not generate proof obligation.

A heuristic strategy has been tried, which seems to provide sensible results; it consists of:

- First, apply the continuity rule until there is no more recursive definitions.
- Then, eliminate initial values. This corresponds to the first step of example 1.
- Finally, deal with the general case.

It should be noted that the initial value elimination step may require unfolding several times the property, depending on the number of initial values to eliminate, until the induction hypothesis can be used. This allows us to tune the number of unfoldings to the needs of the property to be proved. This is, for instance, the case when trying to prove the property:

Example 2 $All(fib \geq 0)$

Here, our initial value elimination strategy provides us with the right number of unfoldings needed to prove the property.

A translation tool, written in Caml, performs the unfolding of the strategy, starting from a Lustre program and a given *All* property, and generates PVS proof obligations. Table 3 displays the resulting PVS proof obligations of example 2. It is easy to see from this example that the proof obligations are obvious and can be discharged with a straightforward PVS strategy.

On the contrary, the strategy fails to provide proof obligations when dealing with non synchronous problems like:

[2] Just think of writing a strategy for automatically obtaining the unfolding of example 1.

```
example2_ex2_node : THEORY
 BEGIN
 IMPORTING streams
fib : VAR stream[int]

ex2_prop1_0 : THEOREM
(    (hd(fib)  >= 0)
) => (1 >= 0)

ex2_prop1_00 : THEOREM
(    (hd(tl(fib))  >= 0)
AND (hd(fib)  >= 0)
) => ((hd(fib)  + 0) >= 0)

ex2_prop1_000 : THEOREM
(    (hd(tl(tl(fib)))  >= 0)
AND (hd(tl(fib))  >= 0)
AND (hd(fib)  >= 0)
) => ((hd(tl(fib))  + hd(fib) ) >= 0)

  END example2_ex2_node
```

Table 3. PVS proof obligations for example 2

Example 3: $All(x \geq 0) \Rightarrow All(x + (x\ when\ c) \geq 0)$

The problem, here, is that the induction hypotheses never allow the remaining goals to be discharged because there is always an extra tl which appears in the goal and not in the hypothesis. In this sense, our strategy looks very much like Wadler's deforestation [20] in the particular context of proof generation.

5 Co-induction

5.1 Processes and Co-inductive Definitions

Processes (co-algebras), (X, D, f, g) are defined by:

- a set of states$X$ and a set of values D;
- a transition function f from X to X;
- an output function g from X to D;

Then, it can be shown [10] that (D^ω, D, hd, tl) is a final co-algebra in the sense that there exists an unique (behavior) function h from X to D^ω such that, for all x in X, $hd(h\ x) = g\ x$ and $tl(h\ x) = h(f\ x)$ where hd, tl are the usual sequence destructors. Equivalently, $h\ x = (g\ x).h(f\ x)$ (Here the uniqueness of h comes simply from the fact that, for any x and n, $h\ x\ n = g(f^n\ x)$).

Such a definition is said co-inductive. Uniqueness gives sense to recursively defined functions, such as $rec\ h : \lambda\ x : (g\ x).h(f\ x)$ provided their co-domains lie within some final co-algebra.

Bisimulation Proofs are based on the observation of the property truth value, i.e., $D = \{false, true\}$. Then an always true property is such that the observation yields the sequence $true^\omega$. If we want to prove that the property holds for some x_0 in X, we must find some property P such that $P(x)$ implies $g(x)$ and $P(f(x))$ and then prove that $P(x_0)$ holds. A "natural" choice for P can be to choose $g^{-1}(true)$.

Infinite Proofs correspond to the `cofix` tactic in Coq and avoid the drawback of having to choose a property. They are based on quite the same rules as induction rules of table 2 but for the ϵ cases which can be discarded. In this sense, co-induction can be seen as simpler than induction.

5.2 The Clocked Stream Case

The idea here, like in [4, 14], is to replace filtered values by an absent value a. Table 4 displays the expression of Lustre primitives in this setting.[3]

$a.xs$ + $a.ys$	$= a.(xs$ + $ys)$
$x.xs$ + $a.ys$	$= a.(x.xs$ + $ys)$
$a.xs$ + $y.ys$	$= a.(xs$ + $y.ys)$
$x.xs$ + $y.ys$	$= (x + y).(xs$ + $ys)$
$a.xs$ `->pre` ys	$= a.(xs$ `->pre` $ys)$
$x.xs$ `->pre` $a.ys$	$= x.(a.ys$ `->pre` $ys)$
$x.xs$ `->pre` $y.ys$	$= x.(y.ys$ `->pre` $ys)$
$a.xs$ `when` $a.cs$	$= a.(xs$ `when` $cs)$
$a.xs$ `when` $c.cs$	$= a.(xs$ `when` $c.cs)$
$x.xs$ `when` $a.cs$	$= a.(x.xs$ `when` $cs)$
$x.xs$ `when` $true.cs$	$= x.(xs$ `when` $cs)$
$x.xs$ `when` $false.cs$	$= a.(xs$ `when` $cs)$
`current` v $a.cs$ $a.xs$	$= a.($`current` v cs $xs)$
`current` v $a.cs$ $x.xs$	$= a.($`current` v cs $x.xs)$
`current` v $false.cs$ xs	$= v.($`current` v cs $xs)$
`current` v $true.cs$ $a.xs$	$= a.($`current` v $true.cs$ $xs)$
`current` v $true.cs$ $x.xs$	$= x.($`current` x cs $xs)$

Table 4. Co-inductive definition of Lustre primitives

Observing properties of such processes leads now to consider D as a three valued set:$\{a, false, true\}$, and we have to decide what is a good observation;

[3] This table could have been made simpler by merging cases. However the given expressions easily distinguish synchronous and non synchronous cases.

we can decide to say that a property is always true if the observation belongs to $\{a, true\}^\omega$. This consists of saying that a property holds as long as its truth value is not $false$. Then the previous proof rules remain valid.

Example 1: We want to prove that the function $current\ true\ c\ x$ is always true as soon as x is always true, i.e. belongs to $\{a, true\}^\omega$.

Here, the bisimulation rule does not work:

- the initial proof is obvious by case analysis because, initially, v and x evaluate to $true$;
- but the induction step does not hold; the observation of the absent value a does not tell us anything on the stored value v that can be observed at the next step and v can be $false$. Notice that cases 2 and 4 could have been eliminated by some clock analysis (they are not synchronous), but the first one is synchronous and cannot be eliminated.

A solution in this case consists of strengthening the property by adding an observation of the stored internal state value v. However, intuitively, we can see that this would not have been necessary if we had used the definitions of section 2.2: here, the observed value always tells us which is the stored value.

Moreover, we can see that the definitions of table 1 are always simpler than the ones of table 4. Thus, infinite proofs are more complex than inductive ones.

5.3 Comparison with Induction

We can show that continuous induction is as powerful a proof principle as co-induction.

Theorem 1. *Continuous induction generates the same proof obligations as bisimulation.*

We have seen that the bisimulation proof of

$$All\ (rec\ h : \lambda\ x : (g\ x).h(f\ x))x_0$$

yields the proof obligations $\dfrac{}{(g\ x_0)}$ and $\dfrac{(g\ x)}{(g\ (f\ x))}$. By coding this property as:

$$All\ (\mathtt{map}(g)\ (rec\ y : x_0.(\mathtt{map}(f)\ y)))$$

and by applying the unfolding strategy of the paper, we easily obtain the same proof obligations.

The comparison with infinite proofs is even easier as both are base on almost the same rules.

6 Related Works

Continuous induction has been used for long in proving programs [18]. Yet, it seems that the first attempt to use it for proving properties on streams goes back to Ashroft and Wadge [3], and our work can be seen as simply an update of this one. However, it is devoted to the Lucid context which was somewhat different from the Kahn-Lustre one. Furthermore the proof of `when` and `current` based programs was not addressed while it constitutes a major motivation here. More recently, Pavlović [19] describes an approach which seems quite similar to ours, but much more generally presented in a categorical framework that makes it quite difficult to understand. Furthermore it does not seem to draw the practical consequences we have drawn from the comparison between continuous induction and co-induction.

7 Conclusion

The problem of dealing with clocks when proving stream programs leads either to consider "absent" elements and apply usual co-induction or to consider both finite and infinite sequences.[4] In the later case, continuous induction provides us with simple and efficient proof schemes and seems a better choice.

Yet, formalizing it within the PVS framework appeared quite inefficient. This is why we chose to run these strategies off-line, in the translation tool itself, 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 is fast and it often finds the right number of unfoldings which allows a property to be proved.

Furthermore, the framework would offer the possibility of dealing with asynchronous and possibly unbounded Kahn networks, and thus the possibility to model and prove mixed synchronous-asynchronous designs. However, in this case, proofs are more difficult to obtain because the generalization step may not converge. Then, this generalization should be provided manually, in the same way as when some invariant strengthening is needed. 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[1], adapted to the synchronous programming context. In this sense, a proof obligation generator like the one described here could be a good starting point.

[4] Co-induction on finite and infinite sequences also requires absent elements.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1995.
2. R. Amadio and S.Coupet-Grimal. Analysis of a guard condition in type theory. In M.Nivat, editor, *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
3. E.A. Ashcroft and W.W. Wadge. Lucid, a formal system for writing and proving programs. *SIAM j. Comp.*, 3:336–354, 1976.
4. 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, 1998.
5. Th. Coquand. Infinite objects in type theory. In *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.
6. Th. Coquand and G. Huet. The calculus of construction. *Information and Computation*, 76(2), 1988.
7. E. Gimenez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs, TYPES'94*, volume 996 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.
8. 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.
9. U. Hensel and B. Jacobs. Coalgebraic theories of sequences in PVS. Technical Report CSI-R9708, Computer Science Institute, University of Nijmegen, 1997.
10. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of EATCS*, 62:229–259, 1997.
11. G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74*. North Holland, 1974.
12. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.
13. P.S. Miner and S.D. Johnson. Verification of an optimized fault-tolerant clock synchronization circuit. In *Designing Correct Circuits*, Electronic Workshops in Computing, Bastad, Sweden, 1996. Springer-Verlag.
14. D. Nowak, J.R. Beauvais, and J.P. Talpin. Co-inductive axiomatization of a synchronous language. In *Theorem Proving in Higher Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 387–399. Springer Verlag, 1998.
15. 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.
16. 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.
17. L. Paulson. *Logic and Computation, Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.
18. D. Pavlović. Guarded induction on final coalgebras. In *Proceedings of the Workshop on Coalgebraic Methods in Computer Science, Lisbon*, volume 11 of *Electronic Notes in Theoretical Computer Science*, 1998.
19. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.