Co-inductive Axiomatization of a Synchronous Language

David Nowak, Jean-René Beauvais, and Jean-Pierre Talpin

IRISA (INRIA Rennes & CNRS)
Campus de Beaulieu, F-35042 Rennes Cédex, France
{*nowak,beauvais,talpin*}*@irisa.fr*

Abstract. Over the last decade, the increasing demand for the validation of safety critical systems lead to the development of domain-specific programming languages (e.g. synchronous languages) and automatic verification tools (e.g. model checkers). Conventionally, the verification of a reactive system is implemented by specifying a discrete model of the system (i.e. a finite-state machine) and then checking this model against temporal properties (e.g. using an automata-based tool). We investigate the use of a theorem prover, Coq, for the specification of infinite state systems and for the verification of co-inductive properties.

1 Introduction

1.1 Motivations

In recent years, the verification of safety critical systems has become an area of increasing importance in computer science because of the constant progression of software developments in sensitive fields like medicine, communication, transportation and (nuclear) energy. The notion of *reactive system* has emerged to concentrate on problems related to the control of interaction and response-time in mission-critical systems. These strong requirements lead to the development of specific programming languages and related verification tools for reactive systems. The verification of a reactive system is done by elaborating a *discrete* model of the system (i.e. as a finite-state machine) specified in a dedicated language (e.g. a synchronous programming language) and then by checking a property (e.g. liveness, dead-lock freedom, etc) against the model (i.e. model checking). *Synchronous* languages (e.g. Esterel [5, 4], Lustre [14], Signal [7, 3, 20], and Argos [17]) have proved to be well adapted to the verification of safety and liveness properties of reactive systems. For instance, model checking has been used at an industrial scale to Signal programs to check properties such as liveness, invariance, reachability and attractivity in [15]. Whereas model checking efficiently decides discrete properties of finite state systems, the use of formal proof systems enables to prove *hybrid properties* about *infinite state systems*. Using a proof system, we can not only prove the safety and liveness of a reactive system but also prove its *correctness* and its *completeness*. Such a proof, of course, cannot be done automatically: it requires human-interaction to direct the strategy.

The prover can nonetheless automate its most tedious and mechanical parts. In general, formal proofs of programs are difficult and time-consuming. We show that, in the particular case of modeling a reactive system using the synchronous language Signal, this difficulty is significantly reduced by the elegant combination between a declarative style of programming and a relational style of modeling.

1.2 Outline

We first briefly introduce Signal and co-induction in Coq. It is not the purpose of this paper to give a complete description of these subjects but just a sight on their principles in order to make the understanding of our contribution easier. Interested readers may find more in [1] about Coq, [11] about co-induction in Coq, and [3] about Signal. Our focus is the definition of a trace semantics for the synchronous language Signal in Coq. We give an example of correctness proof derived from our theorem library about Signal programs.

2 Specifying Reactive Systems with Signal

Synchronous languages like Esterel, Lustre, Signal, or Argos assume that computation takes no time. In reality, it means that computation duration is negligible in comparison with reaction time of the system. This *synchronous* hypothesis is particularly well adapted to verify safety and some forms of liveness.

Signal is a synchronous, declarative, data-flow oriented programming language. It is built around a simple paradigm: a process is a system of equations on signals; and a minimal kernel of *primitives processes*. A signal represents an infinite flow of data. At every instant, it can be absent or present with a value. The instants where values are present are determined by its associated clock.

The primitive processes are introduced in Fig. 1. The symbol := defines an equality between a signal and an expression. It is not an assignment. *Instantaneous relations* are used to specify relations between signals that must be verified at each instant. Hence, the signals involved in an instantaneous relation must be synchronous i.e. at an instant, they must either be all absent or all present. The when operator is used to select some values of a signal according to a boolean condition. x when y is the *down-sampling* of the signal x when y is present and true. *Deterministic merge* of two signal is done by the default operator (with priority to the left signal). It is possible to access to the previous value (*delay*) of a signal x with $x\$$ init v (v is the initial value). The equation $y := x\$$ init v implies that x and y are synchronous. *Parallel composition* is the union of two systems of equations. *Restriction* enables to declare local signals.

The Signal compiler analyses the consistency of the system of equations. It determines whether the synchronization constraints between the signals can be satisfied or not. It determines whether the causal relations between the signals do not form a cycle (i.e. are deadlock free). The Signal compiler then automatically produces executable code in C, Fortran, Ada, or VHDL.

v	value	$P ::= R(x_1, \cdots, x_n)$	instantaneous relation
x, x_i, y, z	signal	$\mid\ z := x$ when y	down-sampling
P, P'	process	$\mid\ z := x$ default y	deterministic merge
R	relation	$\mid\ y := x\$$ init v	delay
		$\mid\ P \mid P'$	parallel composition
		$\mid\ P/x$	restriction

Fig. 1. Signal-kernel

x	-1 2 6 3 -5 12 7 -3 -8 13 ...
$y := x + 1$	0 3 7 4 -4 13 8 -2 -7 14 ...
$zy := y\$$ init 0	0 0 3 7 4 -4 13 8 -2 -7 ...
$py := zy$ when $zy > 0$	$\perp$ $\perp$ 3 7 4 $\perp$ 13 8 $\perp$ $\perp$ ...
$z := py$ default (0 when (event x))	0 0 3 7 4 0 13 8 0 0 ...

Table 1. example of traces

The Table 1 illustrates each of the primitives with a trace. The symbol $\perp$ denotes the absence of a signal.

The rest of the language is built upon the above kernel. Derived operators are defined from the primitive operators, providing programming comfort. E.g., synchro$\{x, y\}$ constrains the signals x and y to be synchronous, i.e. their clocks to be equal. The process $y :=$ event x gives the clock of y i.e. if x is present with any value then y is present and true else y is absent. The process $y :=$ when x gives the clock y of occurrences of the boolean signal x at the value *true* i.e. if x is present with the value *true* then y is present and true else y is absent. The process $z := x$ cell y memorizes values of x and outputs them when y is true. Delays can be made of n instants, or on windows of n past values. Arrays of signals and of processes are available as well.

Example We design a counter modulo n (This kind of counter is useful to design a watch [8]). This process[1] has a constant parameter `n`. It has two input signals `top_sortie` and `top_incr` which are respectively present when the counter value is required and when the counter value must be incremented. These two signals do not have values. We say that they are of type *event* which is a subtype of *bool* i.e. that they can only be absent or present with the value *true*. The process also has two output signals `cpt` (the value of the counter) and `raz` (the event which is present when the counter is reset to 0). In Signal, we write:

```
process mod_counter =
  {integer n}
```

[1] In Signal, a reactive system can be designed modularly as a set of processes. The keyword `process` associates a name and an interface to a set of equations.

```
(? event top_sortie, top_incr
 ! integer cpt; event raz)
```

The counter must be incremented when the signal top_incr is present, or else it keeps its old value (called zcpt):

```
(| zcpt := cpt$ init 0
 |  cpt := (zcpt+1) mod n when top_incr
        default zcpt
```

The counter value must be computed at each tick of top_sortie and top_incr:

```
 | synchro{cpt, top_sortie default top_incr}
```

The signal raz must be present when top_incr is present and cpt is equal to 0:

```
 | raz := when cpt=0 when top_incr |)/zcpt
```

The compiler automatically verifies these equations and produce executable code. This Signal specification is very similar to the specification in natural language.

Example It is not always so easy to specify a reactive system in Signal. For example, the figure 2 is a general purpose counter which is supposed to count from an initial parameter n up to infinity. The output y is the infinite sequence of integers starting at n+1. The frequency of the output y is given as an input signal x. Each time x is present (provided from the environment), the next value of the counter is instantaneously output (signal y). This specification cannot be directly written in Signal. It is expressed saying that x and y are synchronous signals (x^=y), and output y is the previous value of y incremented by one.

```
process counter = {integer n}(? integer x ! integer y)
(|   x ^= y
 |  zy := y$ init n
 |   y := zy+1 |)/zy
```

Fig. 2. A counter in Signal

How can we verify that the program Fig. 2 meets the informal specification "The infinite sequence of integers starting at n+1 up to infinity"? Obviously, this can not be done using model checking. This paper presents an axiomatization which enables to prove this kind of stream specification.

3 Using Co-Induction in Coq

Coq [1] is a proof assistant for higher-order logic. It allows the development of computer programs that are consistent with their formal specification. The logical language used in Coq is a variety of type theory, the *Calculus of Inductive Constructions* [23]. It has recently been extended with *co-inductive types* [11] to handle infinite objects and is thus well suited to represent signals.

3.1 Relation to previous work

As Signal handles infinite flows of data, we face the problem of representing and manipulating infinite objects: traces of signals. A first solution, consists of viewing signals as infinite sequences. In this setting, a signal is represented by a function which associates any instant i (a natural number) with the value v of the signal (if it is present) or with $\perp$ (if it is absent). This solution is used in [2] to handle Lustre programs in PVS and in [12] and [13] to handle Silage programs in HOL. The declarative and equational style of Signal is similar to Lustre. However, Lustre programs always have a unique reference of logical time: they are *endochronous*. Signal specifications differ from Lustre programs in that they can be *exochronous* (i.e they can have many references of logical time). For example, the process `x:=1 | y:=2` does not constrain the clocks of `x` and `y` to be equal. Hence, had we used functions over infinite sequences to represent signals, we would have faced the burden of having to manipulate several, possibly unrelated, indexes of time i; but also the problem of having no higher-order unification available from Coq.

In [21], a circuit is represented as a function from the stream of inputs to the stream of outputs. By contrast, in Signal, a circuit is represented as a set of relations between the streams of inputs and the streams of outputs. We cannot define primitive processes as stream functions because Signal is a declarative language.

For the above reasons, we chose to view the infinite traces of signals as *co-inductive types* [11] and Signal programs as *co-inductive relations*. In [10] and [9], co-inductive types are used to verify reactive systems encoded in CBS (Calculus of Broadcasting Systems) [22]. Within Coq, this model allows to develop both proofs of co-inductive properties and also proofs of inductive properties of signals, as usual. The combined use of induction and co-induction enriches the expressive power of checkable properties.

4 Co-Inductive Definition of Signals

A signal x is defined as a stream of $\perp$ and values v. The dot is the constructor of streams.

$$x ::= (\perp | v).x$$

In the sequel of this paper, we will need to prove stream equality co-inductively. The definitional equality of streams is not sufficient. We expect that two streams differently defined but with the same elements are equal. As in [9], we use extensional equality. The extensional equality predicate **EqSt** is the largest relation verifying the following axiom:

$$(\forall s_1)(\forall s_2)\mathsf{hd}(s_1) = \mathsf{hd}(s_2) \wedge \mathbf{EqSt}(\mathsf{tl}(s_1), \mathsf{tl}(s_2)) \Rightarrow \mathbf{EqSt}(s_1, s_2)$$

And we add the following extensionality axiom:

$$(\forall s_1)(\forall s_2)\mathbf{EqSt}(s_1, s_2) \Rightarrow s_1 = s_2$$

5 Co-Inductive Definitions of Primitive Processes

Let us recall that a primitive process is not a function but only a relation between signals. This is why every primitive process is denoted by a co-inductive predicate which is the largest relation verifying a list of axioms. Practically, the difference from an inductive definition, is that it is possible to use infinitely many axioms from co-inductive definitions.

The parallel composition is denoted by the logical *and* of the underlying logic and the restriction is denoted by an existential quantifier.

Instantaneous Relation. The relation R^n_P is used to specify an instantaneous relation between n signals. At each instant, these signals verify the inductive predicate P. For all inductive predicate P, for all $n \in I\!N$, R^n_P is the largest relation verifying these axioms:

$$\begin{array}{lll}
\mathrm{R}_1: & (\forall x_i)_{i=1,\cdots,n} & R^n_P(x_1,\cdots,x_n) \Rightarrow R^n_P(\perp.x_1,\cdots,\perp.x_n) \\
\mathrm{R}_2: & (\forall x_i)_{i=1,\cdots,n}(\forall v_i)_{i=1,\cdots,n} & \left\{\begin{array}{l} R^n_P(x_1,\cdots,x_n) \\ P(v_1,\cdots,v_n) \end{array}\right. \Rightarrow R^n_P(v_1.x_1,\cdots,v_n.x_n)
\end{array}$$

Down-Sampling. $\mathbf{When}(x, y, z)$ means that z down-sample x when x is present and y is present with the value *true*. **When** is the largest relation verifying the following axioms:

$$\begin{array}{lll}
\mathrm{W}_1: & (\forall x)(\forall y)(\forall z) & \mathbf{When}(x,y,z) \Rightarrow \mathbf{When}(\perp.x,\perp.y,\perp.z) \\
\mathrm{W}_2: & (\forall x)(\forall y)(\forall z)(\forall b) & \mathbf{When}(x,y,z) \Rightarrow \mathbf{When}(\perp.x,b.y,\perp.z) \\
\mathrm{W}_3: & (\forall x)(\forall y)(\forall z)(\forall v) & \mathbf{When}(x,y,z) \Rightarrow \mathbf{When}(v.x,\perp.y,\perp.z) \\
\mathrm{W}_4: & (\forall x)(\forall y)(\forall z)(\forall v) & \mathbf{When}(x,y,z) \Rightarrow \mathbf{When}(v.x,\mathit{false}.y,\perp.z) \\
\mathrm{W}_5: & (\forall x)(\forall y)(\forall z)(\forall v) & \mathbf{When}(x,y,z) \Rightarrow \mathbf{When}(v.x,\mathit{true}.y,v.z)
\end{array}$$

Deterministic Merge. $\mathbf{Default}(x, y, z)$ means that x and y are merged in z with the priority to x. **Default** is the largest relation verifying:

$$\begin{array}{lll}
\mathrm{D}_1: & (\forall x)(\forall y)(\forall z) & \mathbf{Default}(x,y,z) \Rightarrow \mathbf{Default}(\perp.x,\perp.y,\perp.z) \\
\mathrm{D}_2: & (\forall x)(\forall y)(\forall z)(\forall v) & \mathbf{Default}(x,y,z) \Rightarrow \mathbf{Default}(\perp.x,v.y,v.z) \\
\mathrm{D}_3: & (\forall x)(\forall y)(\forall z)(\forall v) & \mathbf{Default}(x,y,z) \Rightarrow \mathbf{Default}(v.x,\perp.y,v.z) \\
\mathrm{D}_4: & (\forall x)(\forall y)(\forall z)(\forall u)(\forall v) & \mathbf{Default}(x,y,z) \Rightarrow \mathbf{Default}(u.x,v.y,u.z)
\end{array}$$

Delay. The co-inductive predicate **Pre** is used to access to the previous value of a signal. **Pre** is the largest relation verifying:

$$\begin{array}{lll} \mathrm{P}_1: & (\forall x)(\forall y)(\forall v) & \mathbf{Pre}(v,x,y) \Rightarrow \mathbf{Pre}(v,\perp.x,\perp.y) \\ \mathrm{P}_2: & (\forall x)(\forall y)(\forall u)(\forall v) & \mathbf{Pre}(v,x,y) \Rightarrow \mathbf{Pre}(u,v.x,u.y) \end{array}$$

The table 2 shows an example of traces verifying the equation $\mathbf{Pre}(v,x,y)$. By definition, x and y must be synchronous. This is why the axiom P_1 states that a $\perp$ in x correspond to a $\perp$ in y. And, informally speaking, P_2 states that if x was present at the previous instant then its value was u and the value of y was the previous stored state v.

x	5	0	$\perp$	9	$\perp$	$\perp$	12	...
y	v	5	$\perp$	0	$\perp$	$\perp$	9	...

Table 2. example of **Pre**

Derived Operators. With the previous defined denotations of primitive processes, we derive the denotations of the derived operators of Signal. **Constant** is used to declare a constant signal. $\mathbf{Constant}(v,x)$ means that at each instant, x is absent or present with the value v. **Constant** is defined by:

$$\mathbf{Constant}(v,x) =_{\mathrm{def}} R^1_{\lambda u.u=v}(x)$$

$\mathbf{Id}(x,y)$ identifies two signals x and y. At each instant, they must be both absent or both present with the same value. **Id** is defined by:

$$\mathbf{Id}(x,y) =_{\mathrm{def}} R^2_{\lambda u.\lambda v.u=v}(x,y)$$

Op is used to apply a binary scalar function at each instant where signals are present. As it is defined with R^3_P, $\mathbf{Op}(o,x,y,z)$ implies that the signals x, y and z are present at the same instants. **Op** is defined by:

$$\mathbf{Op}(o,x,y,z) =_{\mathrm{def}} R^3_{\lambda u.\lambda v.\lambda w.w=o(u,v)}(x,y,z)$$

It is possible to manipulate the clock of a signal (i.e. the instants where it is present) with **Event**. $\mathbf{Event}(x,y)$ means that y is the clock of x. A clock is represented as a signal which can only be absent or present with *true*. **Event** is defined by:

$$\mathbf{Event}(x,y) =_{\mathrm{def}} (\mathbf{Op}(\lambda u.\lambda v.true,x,x,y)$$

It is also possible to constrain two signals to have the same clock. **Synchro** is defined by:

$$\begin{array}{l} \mathbf{Synchro}(x,y) =_{\mathrm{def}} (\exists cx)(\exists cy)(\exists z) \\ \qquad \mathbf{Event}(x,cx) \wedge \mathbf{Event}(y,cy) \wedge \mathbf{Op}(=,cx,cy,z) \end{array}$$

Example The denotation of the process *Counter* is:

$$\begin{array}{l}(\forall v)(\forall x)(\forall y)(\exists zy)(\exists one) \\ \qquad (\mathbf{Synchro}(x,y) \wedge \\ \qquad \mathbf{Pre}(v,y,zy) \wedge \\ \qquad \mathbf{Constant}(1,one) \wedge \\ \qquad \mathbf{Op}(plus,zy,one,y))\end{array}$$

It is only a syntactic transformation from the Signal syntax to the Coq syntax that could be automated.

6 Clock Calculus

In order to infer the clock properties of primitive processes, we first define some clock operators co-inductively.

We define co-inductively the function $\widehat{.}$ which extract the clock of a signal. It is the greatest fixpoint of the following functor F:

$$F(f) = \begin{cases} \bot.x \longmapsto \bot.f(x) \\ v.x \longmapsto true.f(x) \end{cases} \qquad \widehat{.} = gfp(F)$$

We define co-inductively the function $[.]$ which extract the *true* instants of a boolean signal. It is the greatest fixpoint of the following functor F:

$$F(f) = \begin{cases} \bot.x & \longmapsto \bot.f(x) \\ false.x & \longmapsto \bot.f(x) \\ true.x & \longmapsto true.f(x) \end{cases} \qquad [.] = gfp(F)$$

We define co-inductively the function $\widehat{*}$ which extract the common instants of two clocks. It is the greatest fixpoint of the following functor F:

$$F(f) = \begin{cases} (\bot.x, \bot.y) & \longmapsto \bot.f(x,y) \\ (\bot.x, true.y) & \longmapsto \bot.f(x,y) \\ (true.x, \bot.y) & \longmapsto \bot.f(x,y) \\ (true.x, true.y) & \longmapsto true.f(x,y) \end{cases} \qquad \widehat{*} = gfp(F)$$

We define co-inductively the function $\widehat{+}$ which extract the union of the instants of two clocks. It is the greatest fixpoint of the following functor F:

$$F(f) = \begin{cases} (\bot.x, \bot.y) & \longmapsto \bot.f(x,y) \\ (\bot.x, true.y) & \longmapsto true.f(x,y) \\ (true.x, \bot.y) & \longmapsto true.f(x,y) \\ (true.x, true.y) & \longmapsto true.f(x,y) \end{cases} \qquad \widehat{+} = gfp(F)$$

With these definitions we can easily prove the following clock properties of primitive processes:

Proposition 1 (Clock calculus). *For all inductive predicate P, for all $n \in \mathbb{N}$:*

$$
\begin{array}{l}
(\forall x_i)_{i=1,\cdots,n}\; R_P^n(x_1,\cdots,x_n) \Rightarrow \widehat{x_1} = \cdots = \widehat{x_n} \\
(\forall x)(\forall y)(\forall v)\; \mathbf{Pre}(v,x,y) \Rightarrow \widehat{x} = \widehat{y} \\
(\forall x)(\forall y)(\forall z)\; \mathbf{When}(x,y,z) \Rightarrow \widehat{z} = \widehat{x} \mathbin{\widehat{*}} [y] \\
(\forall x)(\forall y)(\forall z)\; \mathbf{Default}(x,y,z) \Rightarrow \widehat{z} = \widehat{x} \mathbin{\widehat{+}} \widehat{y}
\end{array}
$$

7 Co-Inductive Properties of Signal Specifications

In the sequel of this paper, every variable is implicitly universally quantified.

7.1 fairness of a signal

An important hypothesis of the synchronous programming model is that a signal is assumed to have the property of being present (with a value) within a finite deadline (a set of instants). In Signal, this property is translated into the assumption that there only exists a finite number of $\perp$ between two values of a signal (the so-called stuttering-robustness property). We formalize this property using the co-inductive predicate *OnlyFiniteAbsent* . This property about an infinite object obviously needs a co-inductive proof and a co-inductive definition of the predicate. To make sure that there is a finite number of $\perp$ we need to mix co-induction with induction. Hence, *OnlyFiniteAbsent* is the largest relation verifying this axiom:

$$\text{OFA}: \quad \mathit{AbsentPrefix}(v,x,y) \wedge \mathit{OnlyFiniteAbsent}(y) \Rightarrow \mathit{OnlyFiniteAbsent}(x)$$

where *AbsentPrefix* is inductively defined. $\mathit{AbsentPrefix}(v,x,y)$ states that x is of the form $\perp^*.v.y$. It is the smallest relation verifying the axioms:

$$
\begin{array}{ll}
\text{AP}_1: & \mathit{AbsentPrefix}(v,x,y) \Rightarrow \mathit{AbsentPrefix}(v,\perp.x,y) \\
\text{AP}_2: & \mathit{AbsentPrefix}(v,v.x,x)
\end{array}
$$

In order to prove the Proposition 3 we need to prove the following lemma.

Lemma 2. $\text{OnlyFiniteAbsent}(\widehat{x}) \Leftrightarrow \text{OnlyFiniteAbsent}(x)$

Proposition 3. $\text{OnlyFiniteAbsent}(x) \wedge \widehat{x} = \widehat{y} \Rightarrow \text{OnlyFiniteAbsent}(y)$

7.2 Equivalence Relation Between Signals

Two signals are equivalents if they provide the same values in the same order. *EqFlot* is the largest relation verifying this axiom:

$$\text{EF}: \; \mathit{AbsentPrefix}(v,x,x') \wedge \mathit{AbsentPrefix}(v,y,y') \wedge \mathit{EqFlot}(x',y') \Rightarrow \mathit{EqFlot}(x,y)$$

Proposition 4. EqFlot *is an equivalence relation.*

7.3 Stream of a fair signal

It would be interesting to write a function which extract the stream of values of a signal i.e. a function which suppress the $\perp$ of a signal. Unfortunately, it is impossible to write this function in Coq. If its arguments x doesn't verify the predicate *OnlyFiniteAbsent* , this function will not terminate because it will have to extract an infinite number of $\perp$ to find the next value. It could lead to an inconsistent theory. We can only define a predicate $\mathit{Stream}(x, f)$ which verify that the stream f is the stream of values of x.

Stream is the largest relation verifying this axiom:

$$\mathrm{F}: \quad \mathit{AbsentPrefix}(v, x, y) \wedge \mathit{Stream}(y, f) \Rightarrow \mathit{Stream}(x, v.f)$$

From these definitions, we can deduce some major properties of *Stream* and *EqFlot* and some relations between them (Prop. 5). A stream of value is unique (s_1). If a signal has a stream of values then there only exists a finite number of $\perp$ between two values (s_2). Two signals with the same stream of values are equivalent (s_3). Two equivalent signals have the same stream of values (s_4). Two equivalent signals with the same clock are equal (s_5). Finally, we prove a fundamental property of the delay (s_6).

Proposition 5 (Stream calculus).

$$\begin{array}{ll}
\mathrm{Stream}(x, f_1) \wedge \mathrm{Stream}(x, f_2) \Rightarrow f_1 = f_2 & (s_1) \\
\mathrm{Stream}(x, f) \Rightarrow \mathrm{OnlyFiniteAbsent}(x) & (s_2) \\
\mathrm{Stream}(x, f) \wedge \mathrm{Stream}(y, f) \Rightarrow \mathrm{EqFlot}(x, y) & (s_3) \\
\mathrm{Stream}(x, f) \wedge \mathrm{EqFlot}(x, y) \Rightarrow \mathrm{Stream}(y, f) & (s_4) \\
\mathrm{EqFlot}(x, y) \wedge \widehat{x} = \widehat{y} \Rightarrow x = y & (s_5) \\
\mathbf{Pre}(v, x, y) \wedge \mathrm{Stream}(x, s) \Rightarrow \mathrm{Stream}(y, v.s) & (s_6)
\end{array}$$

8 Properties of derived processes

We define co-inductively the function *constant* which compute the infinite stream of a given value. It is the greatest fixpoint of the following functor F:

$$F(f) = v \longmapsto v.f(v) \qquad \mathit{constant} = \mathit{gfp}(F)$$

To make the correctness proofs of processes easier, it is useful to prove the following properties of the derived operators. The stream of a signal x defined by $\mathbf{Constant}(v, x)$ is $\mathit{constant}(v)$ (d_1). Two identified signals have the same stream (d_2). If the stream of the signal x is f_1 and the stream of the signal y is f_2 then the stream of the signal z defined by $\mathbf{Op}(o, x, y, z)$ is the sequence of applications of the function o to each pair of values taken from f_1 and f_2 (d_3). Two signals x and y synchronized by $\mathbf{Synchro}(x, y)$ have the same clock (d_4).

Proposition 6 (Derived processes).

$$
\begin{array}{ll}
\text{OnlyFiniteAbsent}(x) \wedge \mathbf{Constant}(v,x) \Rightarrow \text{Stream}(x, constant(v)) & (d_1) \\
\mathbf{Id}(x,y) \wedge \text{Stream}(x,s) \Rightarrow \text{Stream}(y,s) & (d_2) \\
\mathbf{Op}(o,x,y,z) \wedge \text{Stream}(x,f_1) \wedge \text{Stream}(y,f_2) \Rightarrow \text{Stream}(z, map(o,f_1,f_2)) & (d_3) \\
\mathbf{Synchro}(x,y) \Rightarrow \widehat{x} = \widehat{y} & (d_4)
\end{array}
$$

9 Correctness Proof of the Counter

An accurate (but informal) correctness property of the process `counter` (Fig. 2) is that (1) the input signal `x` and the output signal `y` are synchronous and that (2) the stream of values of `y` is the infinite sequence of integers starting from `n+1`. Using our library of definitions and theorems, we can easily formalize this informal specification (see Theorems 7 and 11).

The following theorem is an immediate application of the proposition (d_4)

Theorem 7. $Counter(n,x,y) \Rightarrow \widehat{x} = \widehat{y}$

To prove the second part of the specification, we need some lemmas. First we study the evolution of *Counter* from one instant to the next instant. Essentially by a case analysis, we prove the two following lemmas.

Lemma 8. $Counter(n, \perp.x, \perp.y) \Rightarrow Counter(n,x,y)$

Lemma 9. $Counter(n, v.x, (n+1).y) \Rightarrow Counter(n+1,x,y)$

Then we study the evolution of *Counter* from one instant to next instant where x and y are present. To prove this lemma, we need the previous lemmas.

Lemma 10. $\text{AbsentPrefix}(v,x,x') \wedge \text{AbsentPrefix}(n+1,y,y') \wedge$
$Counter(n,x,y) \Rightarrow Counter(n{+}1,x',y')$

We define co-inductively the function $from$ which compute the infinite stream of integers starting at a given number. It is the greatest fixpoint of the following functor F:

$$F(f) = n \longmapsto n.f(n+1) \qquad from = gfp(F)$$

Finally we can prove the second part of the correctness property.

Theorem 11.
$\text{OnlyFiniteAbsent}(x) \wedge Counter(n,x,y) \Rightarrow \text{Stream}(y, from(n+1))$

10 Implementation

The above theory has been implemented with Coq using co-inductive types. To prove the correctness of a Signal program, many propositions about primitive processes are needed. We cannot expose them entirely in this paper. Interested readers may find a complete Coq theory with proofs in [19].

The combined use of induction and co-induction enriches the expressive power of checkable properties. In particular, the checking might be used within Coq by simply using primitives tactics: the **Case** tactic expands all the definitions of the signals into their different possible values (e.g. *true*, *false*, $\perp$ for a boolean signal) and the **Auto** tactic then checks the subgoals generated. To make co-inductive proofs, we used the **Cofix** tactic which introduces the current goal as an hypothesis in the context. The goal must be a co-inductive property and the application of this co-inductive hypothesis must be guarded. We also used intensively the inversion tactics [6].

11 Conclusion

An axiomatization of the trace semantics of Signal within a proof assistant like Coq introduces a novel approach for the validation of reactive systems. The Coq tool being continuously updated with new general-purpose proof tactics will benefit Signal program verification. We chose to use co-inductive features of Coq because we found it was the most natural and simplest way to handle infinite objects. Our practice confirmed that this was also an efficient way to prove correctness properties of reactive systems specified in Signal.

We plan to develop a reference Signal compiler in O'Caml [16] and to prove it with Coq. It will automatically translate the Signal syntax into the Coq syntax. Using our co-inductive theorem library, it will enable the interactive proof of, for instance, some clock assumptions that cannot be proved automatically by the compiler (for instance, clocks that depend on arithmetic relations).

Acknowledgments The authors wish to thank Eduardo Giménez for the explanations he provided about the use of co-inductive types in Coq.

This work was partly funded by INRIA, "*action incitative – réécriture dans les systèmes synchrones et asynchrones*".

References

1. Bruno Barras and al. The Coq Proof Assistant Reference Manual. Technical report, INRIA, 1996.
2. Saddek Bensalem, Paul Caspi, and Catherine Parent-Vigouroux. Handling data-flow programs in PVS. Research report (draft), Verimag, May 1996.
3. Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations : the Signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
4. G. Berry. The Constructive Semantics of Pure ESTEREL. Book in preparation, current version 2.0, http://zenon.inria.fr/meije/esterel.
5. Gerard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.
6. C. Cornes and D. Terrasse. Inverting inductive predicates in Coq. In *BRA Workshop on Types for Proofs and Programs (TYPES'95)*, volume 1158 of *LNCS*, 1996.

7. Thierry Gautier, Paul Le Guernic, and François Dupont. SIGNAL v4 : manuel de référence (version préliminaire). Publication interne 832, IRISA, June 1994.
8. Thierry Gautier, Paul Le Guernic, and Olivier Maffeis. For a New Real-Time Methodology. Research report, INRIA, 1994.
9. Eduardo Giménez. An Application of Co-Inductive Types in Coq: Verification of the Alternating Bit Protocol. In *Proceedings of the 1995 Workshop on Types for Proofs and Programs*, number 1158 in LNCS, pages 135–152. Springer Verlag, 1995.
10. Eduardo Giménez. Types Co-Inductifs et Vérification de Systèmes Réactifs dans Coq. In *Proceedings of the Journées du GDR Programmation, Grenoble*, 1995.
11. Eduardo Giménez. *Un Calcul de Constructions Infinies et son Application à la Vérification des Systémes Communicants.* PhD thesis, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, December 1996.
12. Andrew D. Gordon. The Formal Definition of a Synchronous Hardware-Description Language in Higher Order Logic. In *Proceedings of the International Conference on Computer Design*, pages 531–534. IEEE Computer Society Press, October 1992.
13. Andrew D. Gordon. A Mechanised Definition of Silage in HOL. Research report 287, University of Cambridge Computer Laboratory, February 1993.
14. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.
15. Michel Le Borgne, Hervé Marchand, Eric Rutten, and Mazen Samaan. Formal Verification of Signal Programs: Application to a Power Transformer Station Controller. In *Proc. of the 5th Int. Conf. on Algebraic Methodology and Software Technology (AMAST'96)*, number 1101 in LNCS, pages 270–285, July 1997.
16. Xavier Leroy. The Objective Caml system, release 1.07. Documentation and users's manual, INRIA, December 1997.
17. F. Maraninchi. The Argos language: Graphical Representation of Automata and Description of Reactive Systems. In *IEEE Workshop on Visual Languages*, oct 1991.
18. Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 16 September 1991.
19. David Nowak. `http://www.irisa.fr/prive/nowak/signal-coq/`. Coq code, IRISA, 1997.
20. David Nowak, Jean-Pierre Talpin, Thierry Gautier, and Paul Le Guernic. An ML-Like Module System for the Synchronous Language SIGNAL. In *Proceedings of European Conference on Parallel Processing (Euro-Par'97)*, number 1300 in LNCS, pages 1244–1253. Springer Verlag, August 1997.
21. Christine Paulin-Mohring. Circuits as streams in Coq : Verification of a sequential multiplier. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs, TYPES'95*, volume 1158 of *Lecture Notes in Computer Science*, 1996.
22. K. V. S. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25(2–3):285–327, December 1995.
23. B. Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris VII, Mai. 1994.