

## Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon Unité de recherche associée au CNRS n°1398

# Co-Inductive Types in Coq: An Experiment with the Alternating Bit Protocol

Eduardo Giménez

June 1995

Research Report N<sup>o</sup> 95-38



#### Ecole Normale Supérieure de Lyon

46 Allée d'Italie, 69364 Lyon Cedex 07, France Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80 Adresse électronique : lip@lip.ens-lyon.fr

### Co-Inductive Types in Coq: An Experiment with the Alternating Bit Protocol

Eduardo Giménez June 1995

#### Abstract

We describe an experience concerning the implementation and use of co-inductive types in the proof editor Coq. Co-inductive types are recursive types which, opposite to inductive ones, may be inhabited by infinite objects. In order to illustrate their use in Coq, we describe an axiomatisation of a calculus of broadcasting systems where recursive processes are represented using infinite objects. This calculus is used for developing a verification proof of the alternating bit protocol.

Keywords: Program Verification, Type Theory, Co-Inductive Types, Communicating Processes

#### Résumé

Dans cet article nous décrivons une expérience concernant l'implantation et l'utilisation de types co-inductifs dans l'environnement de preuves Coq. Les types co-inductifs sont des types recursifs qui, à la différence des types inductifs, peuvent être habités par des objets infinis. Pour illustrer leur utilisation dans Coq nous décrivons comment axiomatiser un calcul de processus qui communiquent par diffusion, où les processus qui ne terminent pas sont représentés à l'aide des objets infinis. Ce calcul est utilisé pour développer une preuve de vérification du protocole du bit alterné.

Mots-clés: Vérification de Programmes, Théorie des Types, Types Co-Inductifs, Processus Communicants

# Co-Inductive Types in Coq: An Experiment with the Alternating Bit Protocol

Eduardo Giménez \*

June 1995

#### Abstract

We describe an experience concerning the implementation and use of co-inductive types in the proof editor Coq. Co-inductive types are recursive types which, opposite to inductive ones, may be inhabited by infinite objects. In order to illustrate their use in Coq, we describe an axiomatisation of a calculus of broadcasting systems where recursive processes are represented using infinite objects. This calculus is used for developing a verification proof of the alternating bit protocol.

#### 1 Introduction

Most of computing systems deal with calculating an output value from certain input data, this value being the whole answer we expect from the computation. In this kind of systems, termination is an essential issue, since it is a necessary condition to obtain the desired result. However, we could be interested not only in the final value yielded by the computation but also in the process of computation itself. This situation commonly arises when there are several agents involved in the computation, and we are interested in the interaction between them. Consider for example a coffee machine. We could say that what we expect from this "computing system" is to transform some input money into output coffee. However, the way followed by the system to yield the value is also relevant, for example we would not like the machine to give the coffee and then wait for the money, nor to wait for the amount corresponding to the whole coffee inside before delivering it all at once. Under this view, termination is no longer essential. On the contrary, we could be interested in ensuring that the interaction between the agents will continue forever. For example, in a mailing network among several machines, one of the main properties we would want to ensure is that it will keep on sending messages forever, since this is one of the reasons of its existence.

Several previous attempts to describe such computing systems have shown that the consideration of infinite mathematical objects could be useful (if not necessary) for modeling non-ending computations and other circular phenomena related with communicating systems [10, 24, 1]. This consideration was the starting point for us to explore possible extensions of type theory (in particular the Calculus of Constructions) to describe and reason about infinite objects, which would enable the application of existing proof editors like the system Coq [7] to the verification of communicating systems. The purpose of this paper is to report an experiment involving infinite objects carried out in a prototype extension of the system Coq, as well as some of the problems raised by the implementation of this prototype. The extension of the Calculus of Construction underlying it has been already described in [8]. Here we are concerned rather with the practical possibilities and limitations of the prototype.

The example chosen is the verification of the alternating bit protocol, a simple communication system in which a process tries to send messages to another one through an unreliable medium. In order to describe the protocol we represented in Coq a variant of CBS [21, 22, 23], a CCS-like calculus for describing broadcasting systems developed by K. Prasad. The complete development is available by FTP at ftp://cri.ens-lyon.fr/pub/COQ/V5.10 as part of the current version of Coq.

<sup>\*</sup>This research was partially supported by ESPRIT Basic Research Action "Types for Proofs and Programs" and by Programme de Recherches Coordonnées and CNRS Groupement de Recherche "Programmation".

The plan of the paper is as follows. In the next section, we present a short tutorial about co-inductive types in Coq, illustrating how infinite objects can be defined, and developing some small proofs about them. In Section 3 we briefly introduce Prasad's Calculus of Broadcasting Systems [23], discussing some alternatives in the representation of processes. In Section 4 we will present the description of the protocol and the main problems raised by its specification and verification. Finally, Section 5 draws the conclusions of the experiment and some future work is suggested.

#### 2 Co-inductive types in Coq

We assume that the reader is rather familiar with inductive types. These types are characterised by their constructors, which can be regarded as the basic methods from which the elements of the type can be built up. It is implicit in the definition of an inductive type that its elements are the result of a finite number of applications of its constructors. Co-inductive types arise from relaxing this implicit condition and admitting an element of the type to be produced by a non-ending but effective process of construction (defined in terms of the basic methods characterising the type). So, we can consider the wider notion of types defined by constructors (let us call them recursive types) and classify them into inductive and co-inductive ones, depending on whether we consider or not non-ending methods as admissible for constructing elements of the type. Note that in both cases we obtain a "closed type", whose all elements are pre-determined in advance (by the constructors). When we know that a is an element of a recursive type (no matter if it is inductive or co-inductive), what we know is that it is the result of applying one of the basic forms of construction allowed for the type. So the more primitive elimination rule for a recursive type is case analysis, i.e. to considering through which constructor it could have been introduced an element of the set. In the case of inductive sets, the additional knowledge that constructors can be applied only a finite number of times provide us with a more powerful elimination rule, say, the principle of induction. This principle is obviously not valid for co-inductive types, since it is just the expression of this extra knowledge attached to these types.

An example of an inductive type is the type of natural numbers, whose constructors are zero and the successor function. We can introduce this type in Coq by the following command

```
Coq < Inductive Set nat := 0: nat | S: nat \Rightarrow nat. nat\_ind is defined nat\_rec is defined nat\_rect is defined nat\_ind is defined
```

and the principles of induction on natural numbers nat\_ind, nat\_rec and nat\_rect corresponding to the sorts Prop, Set and Type are automatically generated by the system. These principles are added to the (basic) form of elimination given by case analysis. We will not go further in what concerns inductive types, since we assume the reader to be rather familiar with these concepts. A description of the different principles of induction available in Coq can be found in [16, 7].

An example of a co-inductive type is the type of infinite sequences made up with elements of type A, also called streams. In Coq, it can be introduced through the following definition:

```
\texttt{Coq} < \texttt{CoInductive Set Stream:= cons:} \quad A \Rightarrow (\texttt{Stream} \ A) \Rightarrow (\texttt{Stream} \ A). Stream \ \text{is defined}
```

As already mentioned, there are no principles of induction for co-inductive sets. The only elimination rule for streams is case analysis. This principle can be used, for example, to define the destructors  $hd: (Stream \ A) \Rightarrow A$  and  $tl: (Stream \ A) \Rightarrow (Stream \ A)$ :

```
\begin{array}{l} \texttt{Coq} < \texttt{Definition} \ hd := [x : (\textit{Stream} \ A)] \texttt{Case} \ x \ \texttt{of} \ [a : A][s : (\textit{Stream} \ A)] a \ \texttt{end}. \\ hd \ \texttt{is} \ \texttt{defined} \\ \texttt{Coq} < \texttt{Definition} \ tl := [x : (\textit{Stream} \ A)] \texttt{Case} \ x \ \texttt{of} \ [a : A][s : (\textit{Stream} \ A)] s \ \texttt{end}. \\ tl \ \texttt{is} \ \texttt{defined} \end{array}
```

The syntax for case expressions may be a little bit cumbersome for those not familiarised with Coq. The expression Case x of  $\vec{G}$  end defines an element of type Q by case analysis on an element x of a recursive type R. In the most general case Q may be a parametrised type depending on the elements of R, and then the object defined by the case expression has type (Q|x). The list of terms  $\vec{G}$  corresponds to the cases of the analysis, and each one is a function to be applied to the arguments of the respective constructor. The reduction rule associated with this expression is the expected one:

Case (cons 
$$a \ s$$
) of  $h$  end  $\Longrightarrow (h \ a \ s)$ 

At this point the reader should have realised that we have left unexplained what is a "non-ending but effective process of construction" of a stream. In the widest sense, a method is a non-ending process of construction if we can eliminate the stream that it introduces, in other words, if we can reduce any case analysis on it. In this sense, the following ways of introducing a stream are not acceptable.

```
zeros = (cons \ 0 \ (tl \ zeros)) filter (cons \ a \ s) = if \ (P \ a) then (cons \ a \ (filter \ s)) else (filter \ s)
```

The former it is not valid since the stream can not be eliminated to obtain its tail. In the latter, a stream is naively defined as the result of erasing from another (arbitrary) stream all the elements which does not verify a certain property P. This does not always makes sense, for example it does not when there is no element in the stream verifying P, in which case we can not eliminate it to obtain its head. On the contrary, the following definitions are acceptable methods for constructing a stream:

```
(from n) = (cons n (from (S n)))
 alter = (cons true (cons false alter)).
```

The first one introduces a stream containing all the natural numbers greater than a given one, and the second the stream which infinitely alternates the booleans true and false.

In general it is not evident to realise when a definition can be accepted or not. However, there is a class of definitions that can be easily recognised as being valid: those where all the recursive calls of the method are done after having explicitly mentioned which is (at least) the first constructor to start building the element, and where no other functions apart from constructors are applied to recursive calls. This class of definitions is usually referred as guarded-by-constructors definitions [3, 8]. The methods from and alter are examples of definitions which are guarded by constructors. The definition of function filter is not, because there is no constructor to guard the recursive call in the else branch. Neither is the one of zeros, since there is a function applied to the recursive call which is not a constructor.

Guarded definitions are exactly the kind of non-ending process of construction which are allowed in Coq. The way of introducing a guarded definition in Coq is using the special command CoFixpoint. This command verifies that the definition introduces an element of a coinductive type, and checks if it is guarded by constructors. If we try to introduce the definitions above, from and alter will be accepted, while zeros and filter will be rejected giving some explanation about why.

```
\texttt{Coq} < \texttt{CoFixpoint from}: \quad \texttt{nat} \Rightarrow (\textit{Stream} \ nat) \ := \ [n : \texttt{nat}] (\textit{cons} \ n \ (\textit{from} \ (\textit{S} \ n))) \, . from is corecursively defined
```

The elimination of a stream introduced by a CoFixpoint definition is done lazily, i.e. its definition can be expanded only when it occurs at the head of an application which is the argument of a case expression. Isolately, it is considered as a canonical expression which is completely evaluated. Thus, the equality  $(from \ n) \equiv (cons \ nat \ n \ (from \ (S \ n)))$  does not hold as definitional one, since both expressions are canonical and not the same. Nevertheless, this equality can be proved in the propositional sense using Leibniz's equality. In Coq, Leibniz's equality corresponds the following inductive type:

```
Inductive Eq [A:Set;x:A] : A \Rightarrow Prop := refl_equal : (Eq A x x).
```

and the notation a = b is just a shorthand for  $(Eq\ A\ a\ b)$   $(A\ is\ inferred\ by\ Coq\ and\ it\ is\ not\ necessary\ to\ name\ it\ explicitly)$ . The version  $\grave{a}\ la\ Leibniz$  of the equality above follows from a general lemma stating that eliminating and then re-introducing a stream yields the same stream.

```
unfold\_Stream: (x:(Stream\ nat))(x = Case\ x\ of\ cons\ end).
```

The proof is immediate from the analysis of the possible cases for x. This analysis transforms the equality to be proven into the trivial one  $(\cos n \ s) = (\cos n \ s)$ , which follows just by reflexivity. The application of this lemma to  $(f \cos n)$  puts this constant at the head of an application which is an argument of a case analysis, forcing its expansion.

```
(unfold\_stream\ (from\ n)) \quad : \quad (from\ n) = \underbrace{\mathsf{Case}\ (from\ n)\ \mathtt{of}\ cons\ \mathtt{end}}_{\equiv (cons\ n\ (from\ (S\ n)))}
```

#### 2.1 Reasoning about infinite objects

The previous example illustrates the kind of reasoning that can be made using the elimination rule for infinite objects. In fact, what can be proven about a stream using only case analysis is just what can be proven unfolding its method of construction a finite number of times. But this is not always sufficient. Consider for example the following method for appending two streams:

Informally speaking, we expect that for all pair of streams  $s_1$  and  $s_2$ , (conc  $s_1$   $s_2$ ) defines the "the same" stream as  $s_1$ , in the sense that if their definitions would be unfolded "up to the infinite", they would yield definitionally equal normal forms. However, no finite unfolding of the definitions gives definitionally equal terms. Their equality can not be proved just using case analysis. The weakness of the elimination principle proposed for infinite objects is in strong contrast with the power provided by the inductive elimination principles, but this is not actually surprising. It just means that we can not expect to prove very interesting things about infinite objects doing finite proofs. To take advantage of infinite objects, we have to consider infinite proofs as well. For example, if we want to catch up the equality between (conc  $s_1$   $s_2$ ) and  $s_1$ , then we have first to introduce the type of the infinite proofs of equality between streams. This is a co-inductive type, whose elements are build up from a unique constructor, requiring a proof of the equality of the heads of the streams, and an (infinite) proof of the equality of their tails.

Now the equality of both streams can be proved introducing an infinite object of type  $(EqStr s_1 \ (conc \ s_1 \ s_2))$  by a CoFixpoint definition.

Instead of giving an explicit definition, we can use the proof editor of Coq to help us in the construction of the proof. A tactic Cofix allows to place a CoFixpoint definition inside a proof. This tactic introduces a variable in the context which has the same type as the current goal, and its application stands for a recursive call in the construction of the proof. Once the proof is finished, the proof term is checked to verify that all recursive calls are guarded by constructors. We could use the proof editor to define a method of construction whose recursion does not make sense, but the system would never accept to include it as part of the theory. This is the easier solution to implement, but for sure it is not the best one. The main reason is that it is not funny for the user to learn that the proof he/she has been working on for three days is finally not accepted because it was started doing a wrong recursive call. It could be argued that if the user is aware about the proof he/she is constructing, most of the time this would never happen. However, since Coq offers the possibility of using several tactics which automatically search for a proof, the user has not full control about the proof term he is building up.

#### 2.2 Intelligent construction of infinite proofs

When looking for solutions better than checking the complete proof term at the end, the first alternative one could imaginate is to extend the guarded condition to deal with incomplete terms, i.e., terms containing "placeholders" standing for those parts of the proof which remain to be constructed. Such an extended condition could be tested after each application of a tactic. If the refined term does not satisfy it, we can consider that the tactic has failed, and make the prover to roll back to the previous state of the proof. In order to extend the condition to incomplete terms, we can reason in this way: each time a placeholder is found during the checking, we just assume that it will be replaced later on by a term satisfying the condition. In other words, the condition is unconditionally satisfied for any placeholder. Since the guard checking is done downwards from the top of the proof, if the term used in the next refinement makes the proof not guarded, certainly the next check will detect the error.

However, this solution would make the proof engine very inefficient. Since the time needed to verify the condition is proportional to the size of the term proof, the further we advance in the construction of the proof, the slower the application of a tactic would be. Some experimental extensions have shown that already in medium-size proofs this checking would make the proof engine slow enough to discharge this possibility.

A way of improving this idea consists in doing the checking in a lazier way. After applying a tactic, instead of verifying the whole proof term from the top, we can just keep at each placeholder the information yielded by the checking so far. In this way, we should check only the term generated by the tactic for the refinement. This presupposes to transform the (boolean) guard condition into a function that, given an incomplete term, associates to each of its placeholders the list of constraints that should be satisfied by a term refining the placeholder. The problem is that this transformation is not as simple as extending the checking to incomplete terms. In some situations the conditions to be attached to a placeholder may depend on those attached to other placeholders. For example, if an incomplete term which is just a placeholder? 1 is just refined with an application (?1 ?2), the restrictions that ?2 should satisfy depend on the term used to refine? 1 If ?1 is a constructor, then ?2 could be a recursive application of the proof, otherwise it can not 1.

The solution implemented in Coq at the moment is a compromise between the two alternatives presented: by default, the proof engine verifies the condition only when the term is completed, but the user can require it to run the checking on an incomplete proof at any moment during the construction of the proof using a special command Guarded.

#### 3 The Calculus of Broadcasting Systems

After this brief introduction to co-inductive types, we turn to the representation of the calculus of processes that will be used to describe the protocol. This calculus is essentially Prasad's Calculus of Broadcasting Systems (CBS) as it is presented in [22], to which we have added the possibility of having several transmission channels as well as failures in the communications.

#### 3.1 Representation of Processes

CBS is a CCS-like calculus for describing communications among several processes running in parallel. The mode of communication is by broadcasting, i.e. when one process speaks on a certain channel, all other processes listening to the same channel can hear. This is a typical situation in communications by waves transmitted through the air, where processes may transmit on different wave frequencies. At a synchronisation point in the communication, each process can choose either to speak or to hear on a certain channel. If several processes speak at the same time, then the data is lost, and no process in the system hear what was said.

After a meeting, the processes evolve deterministically. The only non determinism lies in the choice they can make between hearing or speaking. We will denote  $c\{a!p|x?q\}$  the process which may choose either to say a on channel c to become the process p, or to hear a value v on channel c to become q[x/v]. So x is considered to be bound in the subexpression x?q. We will also admit that a process may remain blocked waiting to hear a value, such process is denoted  $c\{x?q\}$ . These are the basic process communicating agents

<sup>&</sup>lt;sup>1</sup> This situation arises for example when the tactic Cut is applied

that are considered in the calculus. Two processes p and q can be launched in parallel, and the result of this operation is also considered as being a process, denoted (p || q). This amounts to assume that a process is capable of starting new processes by itself. From this informal explanation, it seems natural to represent the type of processes by the following co-inductive type:<sup>2</sup>

```
Section Process. 

Variable Channel: Set. 

Variable A: Set. 

CoInductive Set Process:= 

TALK: Channel \Rightarrow A \Rightarrow Process \Rightarrow (A \Rightarrow Process) \Rightarrow Process \mid 

LISTEN: Channel \Rightarrow (A \Rightarrow Process) \Rightarrow Process \mid 

PAR: Process \Rightarrow Process \Rightarrow Process.
```

Remark that in this representation a (basic) process is like a tree representing all the possible situations it could run into. For this reason the type is co-inductive, since we do not want to exclude a priori the possibility for a process to evolve infinitely. If we want to describe processes that may have a finite life, we could add another constructor NIL representing a dead process.

**Example 3.1** The following process tries to repeat all the values said by other processes on certain channel  $c_1$  into a different channel  $c_2$ .

```
CoFixpoint HEAR: Process:= (LISTEN c_1 [x: A](REPEAT x)) with REPEAT: A \Rightarrow Process := [x:A](TALK <math>c_2 x HEAR [y:A](REPEAT y))).
```

The set of channels and the type of the values that can be sent through them are general parameters of the definition<sup>3</sup>. Remark that this is a naive solution, because if the processes sending the values on channel  $c_1$  speak faster than REPEAT does it on channel  $c_2$ , then some of the messages will not be repeated.  $\bullet$ 

The processes originally considered in [22] did not have channels. Instead of this, an extra constructor of processes was considered, which allows to enclose several process speaking the same language A into a box. A box can be put in parallel with other boxes speaking a different language B, provided that two functions for translating from one language to the other are given. This would be an alternative solution to the problem of restricting the language spoken on a channel, since all the process inside a box could be considered as connected through the same one. This choice can be reflected introducing the following set former:

```
CoInductive Process: Set \Rightarrow Set := 
TALK: \forall A : Set. \ A \Rightarrow (Process \ A) \Rightarrow (A \Rightarrow (Process \ A)) \Rightarrow Process \ |
LISTEN: \forall A : Set. \ (A \Rightarrow (Process \ A)) \Rightarrow (Process \ A) \ |
PAR: \forall A : Set. \ (Process \ A) \Rightarrow (Process \ A) \Rightarrow (Process \ A) \ |
TRANS: \forall A, B : Set. \ (A \Rightarrow B) \Rightarrow (B \Rightarrow A) \Rightarrow (Process \ B) \Rightarrow (Process \ A).
```

Note that under this view, the general parameter A in the definition of the type of processes becomes an index, because Process occurs applied to a variable different from A in the type of TRANS. This definition is perfectly admissible in Coq but, as it will be shown in section 3.2, the choice of indexing the type by the channels would lead us to some annoying situations during the proof. We therefore prefer to keep the first representation proposed, using channels instead of the constructor TRANS.

<sup>&</sup>lt;sup>2</sup>In Coq, the Variable declarations corresponds to the introduction of general parameters relative to all the definitions after it in the scope of the current Section, somewhat like the expression let A be a set ... frequently used in mathematical text books. After the section is closed, these parameters are abstracted away from the definitions.

 $<sup>^3</sup>$  We will refer sometimes to the parameter A as the "language" spoken by the process.

#### 3.2 Operational Semantics

The intuitive semantics suggested in the previous section can be formally described introducing for each channel a transition relation  $P \xrightarrow{w}_c Q$  between processes labeled by an action w and a channel c. The possible actions are the transmission or the reception of certain value on the channel. We will write !v the action of transmitting and ?v the action of receiving certain value v:A. Figure 1 lists the rules defining the transition relation on a reliable channel. The rules for the basic agents are straightforward, just note that processes do not care about reception actions on those channels different from the one they are listening to. To obtain a transmission from two processes in parallel it is necessary that one speaks and the other listens. To make two processes in parallel hear a value each one must hear the value. The communication is perfect: only one process speaks at a time and all messages reach the receivers.

$c\{x?Q\} \xrightarrow{?v}_c (Q \ v)$		$c'\{a!P x?Q\} \xrightarrow{?v}_c c'\{a!P x?Q\}, c \neq c'$	
$c\{a!P x?Q\} \xrightarrow{:a}_{c} P$	$c\{a!P x$	$c?Q\} \xrightarrow{?v}_c (Q \ v)  c'\{c\}$	$\{x?Q\} \xrightarrow{?v}_{c} c'\{x?Q\}, c \neq c'$
$P \xrightarrow{!\mathbf{v}}_{c} P' \ Q \xrightarrow{?\mathbf{v}}_{c} Q'$	$\rightarrow_c P' Q \xrightarrow{?v}_c Q'$		$P \xrightarrow{?\mathbf{v}}_{c} P' \ Q \xrightarrow{?\mathbf{v}}_{c} Q'$
$(P \mid\mid Q) \xrightarrow{:_{\mathbf{v}}}_{c} (P' \mid\mid Q')$	(1	$P \mid\mid Q) \xrightarrow{:v}_{c} (P' \mid\mid Q')$	$ (P \mid\mid Q) \xrightarrow{?\mathbf{v}}_{c} (P' \mid\mid Q') $

Figure 1: Communication on reliable channels

Let us consider now that the medium is such that a value sent through the channel may be lost without the sender realises about this fact. It is possible to model this situation by lifting the type of values transmited with a special value  $\tau$  representing "noise", a failure in the communication. A transmission action can now be either transmitting a value, as before, or transmitting noise. To give meaning to the action of transmitting noise, the rules in Figure 1 are extended with the rules on Figure 2. These rules specify that a talking process may loose its value without it realises about this fact, and that in order to obtain a loosy transmission from two processes in parallel, at least one of them must lost its message.

$c\{a!P x?Q\} \xrightarrow{\cdot r}_{c} P$					
$P \xrightarrow{!\tau}_c P'$	$Q \xrightarrow{\cdot ! \tau}_c Q'$	$P \xrightarrow{!\tau}_c P' \ Q \xrightarrow{!\tau}_c Q'$			
$(P \mid\mid Q) \xrightarrow{:\tau}_{c} (P' \mid\mid Q)$	$P \parallel Q) \xrightarrow{!\tau}_{c} (P \parallel Q')$	$P \parallel Q) \xrightarrow{:\tau}_{c} (P' \parallel Q')$			

Figure 2: Communication on unreliable channels

Describing these transition relations in Coq as inductive types is just routine. Let us start introducing the type of messages lifted with the special value  $\tau$  and the type of actions.

```
 \textbf{Inductive } Set \ Signal := Noise : \ Signal \mid \ Clear : \ A \Rightarrow Signal.
```

Inductive Set Action := Transmit : Signal  $\Rightarrow$  Action | Receive :  $A \Rightarrow$  Action.

The transition relation for reliable and unreliable channels are introduced as inductive predicates parametrised by the channel c on which the action is produced. Both relations are inductive, which means that a transmission is meaningful only for those systems in which there is a finite number of processes running in parallel. For the sake of space we do not present here all their introduction rules but just some of them.

```
\label{eq:continuous} \textbf{Inductive } SafeProcTrans \ [c:Channel] : Process \Rightarrow Action \Rightarrow Process \Rightarrow Set
```

```
 \begin{array}{l} := \ sttalk_1 : \ \forall f : A \Rightarrow Process. \ \ \forall v : A. \ \ \forall p : Process. \\ (SafeProcTrans c \ (Talk \ c \ v \ p \ f) \ !v \ p) \ | \\ \dots \\ stpar_3 : \ \forall p_1, p_2, q_1, q_2 : Process. \ \forall v : A. \\ (SafeProcTrans c \ p_1 \ ?v \ p_2) \Rightarrow \\ (SafeProcTrans c \ q_1 \ ?v \ q_2) \Rightarrow \\ (SafeProcTrans c \ (Par \ p_1 \ q_1) \ ?v \ (Par \ p_2 \ q_2)) \\ \end{array}
```

```
 \begin{array}{l} \text{Inductive } \textit{UnrelProcTrans} \ [\textit{c} : \textit{Channel}] : \textit{Process} \Rightarrow \textit{Action} \Rightarrow \textit{Process} \Rightarrow \textit{Set} \\ := \dots \\ \\ \textit{utpar}_1 : \forall p_1, p_2, q_1, q_2 : \textit{Process}. \ \forall \textit{v} : \textit{A}. \\ & (\textit{UnrelProcTrans} \ \textit{c} \ p_1 \ !\textit{v} \ p_2) \Rightarrow \\ & (\textit{UnrelProcTrans} \ \textit{c} \ q_1 \ ?\textit{v} \ q_2) \Rightarrow \\ & (\textit{UnrelProcTrans} \ \textit{c} \ (\text{PAR} \ p_1 \ q_1) \ !\textit{v} \ (\text{PAR} \ p_2 \ q_2)) \mid \dots \\ \\ \text{utpar}_7 : \forall p_1, p_2, q_1, q_2 : \textit{Process}. \ \forall \textit{v} : \textit{A}. \\ & (\textit{UnrelProcTrans} \ \textit{c} \ p_1 \ !\textit{\tau} \ p_2) \Rightarrow \\ & (\textit{UnrelProcTrans} \ \textit{c} \ q_1 \ !\textit{\tau} \ q_2) \Rightarrow \\ & (\textit{UnrelProcTrans} \ \textit{c} \ q_1 \ !\textit{\tau} \ q_2) \Rightarrow \\ & (\textit{UnrelProcTrans} \ \textit{c} \ (\text{PAR} \ p_1 \ q_1) \ !\textit{\tau} \ (\text{PAR} \ p_2 \ q_2)) \\ \end{aligned}
```

Let us fix some useful terminology. A trace from p is a non-ending transition path of transmission actions

$$p \xrightarrow{\stackrel{\nabla_0}{|w_0|}}_{c_0} p_0 \xrightarrow{\stackrel{\nabla_1}{|w_1|}}_{c_1} p_1 \xrightarrow{\stackrel{\nabla_2}{|w_2|}}_{c_2} p_2 \dots$$

where the transitions are labeled not only with actions but also with proofs  $\nabla_0, \nabla_1, \ldots$  of the transitions. So a trace represents a possible future for p, i.e,. a way in which things may happen. A discourse is a stream of signals  $w_0, w_1, \ldots$  labeling the transmissions of a trace, and describes the conversation among the processes that an external observer would hear.

**Example 3.2** Transition proofs can be also used to describe properties about the communications. For example, consider a communicating system  $(p \mid\mid q)$  in which two process are running in parallel. The property "p succeeds in sending its message" can be introduced as the following inductive predicate on transition proofs:

```
Inductive LeftTalksSafely [c: Channel]

: (p: Process)(w: Action)(q: Process)(UnrelProcTrans\ c\ p\ w\ q) \Rightarrow Prop:=
lts: \forall p_1, p_2, q_1, q_2: Process. \forall v: A.
\forall H_1: (UnrelProcTrans\ c\ p_1: v\ p_2)).
\forall H_2: (UnrelProcTrans\ c\ q_1: v\ q_2)).
(LeftTalksSafely c\ (PAR\ p_1\ q_1): v\ (PAR\ p_2\ q_2)\ (utpar_1\ p_1\ p_2\ q_1\ q_2\ v\ H_1\ H_2)).
```

Another way of specifying this predicate is as a propositional function defined by case analysis on the transition proof, which associates the absurd proposition False to all the proofs except to those built with  $atpar_1$ . If this way is followed, the constructor its above can be proved as a property of the predicate.

#### 3.3 Determination properties.

Before turning our attention to the verification of the alternating bit protocol, we introduce some properties about communicating processes which will be used in the proof. Let us start assuming a function which associates to each channel c of the communicating system a certain labeled transition relation  $p \xrightarrow{w}_{c} q$ .

```
Section Determination Properties. Variable \longrightarrow: Channel \Rightarrow Process \Rightarrow Action \Rightarrow Process \Rightarrow Set.
```

1. Simple Determination. When studying a communicating process, we are most of the time interested in studying its evolution if we constraint the future in certain way, i.e., we are interested not in all possible traces of the process but just in those where some property about the communications is sooner or later determined to happen. Examples of this kind of "hypothesis about the future" that could be interesting are : there will be a clear transmission, or someone will eventually talk on channel c, or we know that in the process  $(p \mid | q)$  the sub-process p will not talk forever. A wide class of these hypothesis can be thought as an instance of the following (second order) property: somewhere in the future, P is determined to fail, where P is a property about the transmissions of the system. In order to characterise this property in an abstract

way, let us imagine a transition tree whose nodes are processes, and where the children of a given node q are all the q' such that  $q \xrightarrow{w}_c q'$  for some action w. Thus, the branches of this tree are all the possible traces from the root. Assume a predicate P such that  $\neg P$  is what is determined to happen.

#### Section Determination.

Variable  $P: \forall c: Channel. \ \forall w: Signal. \ \forall p,q: Process. \ p \xrightarrow{\pi}_{c} \ q \Rightarrow Prop.$ 

The property P determined to fail can be thought as a property about the future of a certain process p, this future being represented by the discourse s that will be heard from it. We define it saying that a proof that P is determined to fail is a proof that the biggest transition tree that can be constructed starting from p, and such that all its transitions verify P, is a well-founded tree. This is expressed the following inductive definition:

```
Inductive Det: \ Process \Rightarrow (Stream \ Signal) \Rightarrow Prop := notyet : \ \forall p : Process. \ \forall s : (Stream \ Signal).
(\exists q_0 : Process. \ \exists c_0 : Channel. \ p^{:(hd\ s)}_{c_0} \ q_0)
\Rightarrow (\ \forall c : Channel. \ \forall q : Process. \ \forall t : p \xrightarrow{\circ}_{c} q.
(P\ c\ (hd\ s)\ p\ q\ t) \Rightarrow (Det\ q\ (tl\ s))\ ) \Rightarrow (Det\ p\ s).
```

Remark that this is an inductive type with only one (recursive) constructor. The only way to complete the construction of an element of it is that any path from the root reaches a process whose all sons are generated from communications not verifying P. When this situation is reached, the proof can be finished by absurdity, since any way of going deeper in the transition tree lead us into a contradiction with the hypothesis that P holds. The constructor also requires that there is at least one son  $q_0$  of the process, so that if we are able of constructing an absurdity, this can be only with respect to the hypothesis that P holds.

2. Cyclic Determination. We could constraint the future even more, imposing not only that the property P is determined to fail, but also that this fact is cyclic, i.e. that it will always re-happen later on. To describe this property we have first to introduce the notion of a witness of the failure of P. A witness of  $\neg P$  from  $p_0$  is a process  $p_n$  accessible from  $p_0$  at which the property P fails to happen for the first time. A proof that  $p_n$  is a witness consists in constructing a finite transition path

$$p_n \ \, c_n^{\stackrel{\nabla n}{\underset{|w|_n}{|w|_n}}} \ \, p_{n-1} \ldots \ \, c_1^{\stackrel{\nabla_1}{\underset{|w|_1}{|w|_1}}} \ \, p_0 \ \, c_0^{\stackrel{\nabla_0}{\underset{|w|_0}{|w|_0}}} \ \, p$$

such that  $(P \ c_i \ w_i \ p_{i-1} \ p_i \ \nabla_i)$  holds for all  $i=1,\ldots n-1$ , and does not hold for i=n. The type of these paths can be defined inductively as follows:

```
Inductive Witness [pn: Process; rs: (Stream Signal)]
: Process \Rightarrow (Stream Signal) \Rightarrow Prop :=
start: \forall p: Process. \forall cn: Channel. \forall wn: Signal. \forall t: p^{\frac{1}{2}wn} \rightarrow_{cn} pn.
(\neg (P \ cn \ wn \ p \ pn \ t)) \Rightarrow (Witness \ pn \ rs \ p \ (cons \ wn \ rs))
| \ goback: \forall p: Process. \forall s: (Stream Signal).
\forall c: Channel. \forall w: Signal. \forall q: Process. \forall t: p^{\frac{1}{2}w} \rightarrow_{c} q.
(P \ c \ w \ p \ q \ t) \Rightarrow (Witness \ pn \ rs \ q \ s)
\Rightarrow (Witness \ pn \ rs \ p \ (cons \ w \ s)).
```

Remark that the path is constructed backwards, starting from the witness  $p_n$  and climbing up in the transition tree until the root is reached. The general parameter rs corresponds to the discourse of  $p_n$ , what remains after erasing the initial segment  $w_1, \ldots w_n$  from the discourse of  $p_0$ .

Once we have introduced this notion, we can say that P is cyclicly determined to fail from p if it is simply determined to fail from p, and it is determined to fail again from any witness q of its failure from p, and again from any witness of its failure from q, and so on forever. To put it in other words, the witnesses of the

failure determine a frontier cutting the tree, such that any node at this frontier is the root of a tree which can be cut again generating a second frontier, all whose nodes are roots of trees that can be cut once more, etc. This image gives rise to the following co-inductive definition:

```
CoInductive CycDet: Process \Rightarrow (Stream\ Signal) \Rightarrow Prop:=
cycdet: \forall p: Process. \ \forall s: (Stream\ Signal). \ (Det\ p\ s) \Rightarrow
(\forall q: Process. \ \forall rs: (Stream\ Signal). \ (Witness\ q\ rs\ p\ s) \Rightarrow (CycDet\ q\ rs))
\Rightarrow (CycDet\ p\ s).
```

Note that we have to require that the property P is actually determined to fail, otherwise we could prove that it re-fails from any witness just by absurdity from the hypothesis that there exists a witness of its failure. Furthermore, it is not enough to require that indeed there exists a witness  $q_0$ . All possible paths must be cut with a failure of P, since the property could never fail on paths different from the one containing  $q_0$ .

Before defining the last determination property, let us introduce here two lemmas about the cyclic determination that will be necessary in the next section for the verification proof. The first one states that if the property P will cyclicly fail from p, and  $p \xrightarrow{w}_{c} q$ , then P will cyclicly fail also from q.

```
Lemma TransPreservation: \forall p : Process. \ \forall ss : (Stream \ Signal). \forall c : Channel. \ \forall w : Signal. \ \forall q : Process. \ p \xrightarrow{\text{'w}}_{c} \ q \Rightarrow (CycDet \ p \ (cons \ w \ ss)) \Rightarrow (CycDet \ q \ ss). Proof ....
End Determination. <sup>4</sup>
```

The only proof we have found for this lemma requires the decidability of the property P. If we assume this, the proof is very simple. When P holds for the communication leading from p to q, we have that (1) P will fail from q since it is determined to fail from p and  $p \xrightarrow{w}_{c} q$  and (2) since P holds for the communication, any witness of its failure from q is also a witness from p. Then, using the hypothesis that P will cyclicly fail from any witness from p, we have that this will also happen from any witness from p. If P does not hold for the communication, then p is a witness of p from p, and then the property will cyclicly fail from it.

The interest of the lemma is just to prove the following theorem, necessary for the next section: if a property P is cyclicly determined to fail from p, and we know a witness q of the failure of another property Q from p, then the property P is also cyclicly determined to fail from q.

```
Section It_Still_Happens.  \begin{aligned} & \text{Variable } P, Q : \forall c : \textit{Channel.} \ \forall w : \textit{Signal.} \ \forall p, q : \textit{Process.} \ p \xrightarrow{!v}_{c} \ q \Rightarrow \textit{Prop.} \end{aligned}   \begin{aligned} & \text{Theorem } Still \text{Happens :} \\ & \forall p : \textit{Process.} \ \forall ss : (Stream \, Signal). \\ & \forall q : \textit{Process.} \ \forall ss_1 : (Stream \, Signal). \\ & (Witness \, Q \, q \, ss_1 \, p \, ss) \Rightarrow (CycDet \, P \, p \, ss) \Rightarrow (CycDet \, P \, q \, ss_1). \end{aligned}   \end{aligned}   \begin{aligned} & \text{Proof ....} \end{aligned}  End It_Still_Happens.
```

The proof follows just by a straightforward induction on the proof that q is a witness, using the previous lemma to propagate P forwards through the transition path which leads to the witness.

3. Serial Determination. A generalisation of the cyclic determination consists in saying that the property P which must fail somewhere in the future is not always the same, but that it varies depending on a stream of values  $b_1, b_2, \ldots$  of certain type B. In other words, that there is in fact a stream of properties

 $<sup>^4</sup>$  We close the section **Determination** after this theorem, which makes the general parameter P to be abstracted away from all the previous definitions.

 $(P \ b_1), \ (P \ b_2), \ldots$ , which will fail one after the other. A slight modification of the previous definition allows us to specify this new property: P is regarded as a function which associates to each  $b_i$  a certain property about the communications, and the stream  $b_1, b_2, \ldots$  of values of certain type B is introduced as an argument of the property.

```
Section A_stream_is_said_on_certain_channel.  
Variable B:Set.  
Variable P:B\Rightarrow \forall c:Channel.\ \forall w:Signal.\ \forall p,q:Process.\ p\xrightarrow{!v}_c q\Rightarrow Prop.  
CoInductive SerDet:(Stream\ B)\Rightarrow Process\Rightarrow (Stream\ Signal)\Rightarrow Prop:=serdet:\ \forall s:(Stream\ B).\ \forall p:Process.\ \forall ss:(Stream\ Signal).\ (Det\ (P\ (hd\ s))\ p\ ss)\Rightarrow (\forall q:Process.\ \forall rss:(Stream\ Signal).\ (Witness\ (P\ (hd\ s))\ q\ rss\ p\ ss)\Rightarrow (SerDet\ (tl\ s)\ q\ rss))\Rightarrow (SerDet\ s\ p\ ss).  
End A_stream_is_said_on_certain_channel.
```

After this property we close the section concerning determination properties about, abstracting also the parametric transition relation  $\longrightarrow$  from all the definitions above.

#### 4 The alternating bit protocol

End Determination\_Properties.

In this section we finally go into the verification of the protocol. Only the main ideas of the proof will be presented, the reader interested in the details may consult the whole proof at the contributions directory available with the system Coq [7].

#### 4.1 Specification

The problem to be solved is the following one: a process has to communicate to another one a stream of messages through an unreliable channel, so that all the values of the stream are correctly received and in the same order they were sent. As we are not interested in what the receiver will do with the received values, we just imagine that it re-send them on a reliable channel to whoever may need them. We also assume that, even though the unreliable channel may loose some values, after a finite number of attempts it is capable of transmitting the value clearly. The solution of this problem is well-known and consists in letting both the sender and the receiver keep a bit which allows to re-synchronise the communication when a message is lost. The sender attaches its current bit to the messages it sends, so that the receiver can compare it with its own bit. Each time the receiver hears a message with the same bit he holds, it is sure that it gets the next message to repeat, and it sends it on the other channel. Then, it sends its current bit as acknowledgment to the sender, which is programmed to repeat the same message until the acknowledgment it receives corresponds to the bit it has. Thus, in this example, the general parameters in the definition of the type of processes are instantiated with the following inductive types:

```
Inductive Set Channel := chnl_1 : Channel | chnl_2 : Channel.
Inductive Set Act := Send : bool \Rightarrow A \Rightarrow Act | Ack : bool \Rightarrow Act | Relay : A \Rightarrow Act.
```

For the sake of readability, these general parameters will be systematically omitted in the rest of the section, writing *Process* instead of (*Process Channel Act*), TALK instead of (TALK Channel Act), etc. This is not just an abbreviation for this paper, but Coq actually provides some mechanisms to hide and infer general parameters [7].

The fact that the channel communicating the sender and the receiver is unreliable but the one on which the receiver repeats the messages is not is modeled respectively associating the transition relations *UnrelProcTrans* and *SafeProcTrans* defined on Section 3.2 to each one. So the transition relation is defined by case analysis on the channel on which the action is performed, as follows:

The following descriptions of the sender and the receiver correspond to a (a slight modification of) the LML programs proposed in [22]. The sender is described using two mutually dependent guarded definitions, which differentiates the state of sending a value for the first time and the state of waiting for an acknowledgment:

```
 \begin{aligned} \text{CoFixpoint SEND} &: bool \Rightarrow (Stream \ A) \Rightarrow Process := \\ [b:bool][s:(Stream \ A)] \\ &\quad (\text{TALK } chnl_1 \ (Send \ b \ (hd \ s)) \ (\text{SENDING } b \ s) \ [v:Act](\text{SEND } b \ s)) \end{aligned}  with SENDING : bool \Rightarrow (Stream \ A) \Rightarrow Process := \\ [b:bool][s:(Stream \ A)] \\ &\quad (\text{TALK } chnl_1 \ (Send \ b \ (hd \ s)) \ (\text{SENDING } b \ s) \\ &\quad [v:Act]\text{Case } v \ \text{of} \\ &\quad [b_1:bool][a:A](\text{SENDING } b \ s) \\ &\quad [b_1:bool]\text{if } (eqbool \ b_1 \ b) \ \text{then } (\text{SEND } (not \ b) \ (tl \ s)) \ \text{else } (\text{SEND } b \ s) \\ &\quad [a:A](\text{SENDING } b \ s) \\ &\quad \text{end } ). \end{aligned}
```

The receiver is also introduced through (three) mutually dependent guarded definitions. Each function of the block respectively corresponds to the state of waiting for the sender's next message, repeating this message on channel two, and sending an acknowledgment for it.

```
Cofixpoint acking: bool \Rightarrow Process := [b:bool]
(LISTEN \ chnl_1 \ [v:Act] \texttt{Case} \ v \ \text{of}
[b_1:bool] [a:A] \texttt{if} \ (eqbool \ b_1 \ (not \ b)) \ \text{then} \ (\texttt{OUT} \ (not \ b) \ a) \ \text{else} \ (\texttt{ACK} \ b)
[b_1:bool] (\texttt{ACKING} \ b)
[a:A] (\texttt{ACKING} \ b)
end)
with \texttt{OUT}: bool \Rightarrow A \Rightarrow Process := [b_1:bool] [a:A] (\texttt{TALK} \ chnl_2 \ (Relay \ a) \ (\texttt{ACK} \ b_1) \ [v:Act] (\texttt{OUT} \ b_1 \ a))
with \texttt{ACK}: bool \Rightarrow Process := [b:bool] (\texttt{TALK} \ chnl_1 \ (Ack \ b) \ (\texttt{ACKING} \ b) \ [v:Act] (\texttt{ACK} \ b)).
```

The description of the protocol just consists in putting the processes SENDING and ACK in parallel, both starting with the same bit. At the beginning there is a synchronisation phase in which the first message of the sender is not taken into consideration by the receiver<sup>5</sup>, so the sender is programmed to send its first message twice.

Definition ABP := [b : bool][s : (Stream A)](PAR (SENDING b) (cons (hd s) s)) (ACK b)).

#### 4.2 A verified CBS simulator

In [22] Prasad proposes an interpreter for this calculus written in the programming language LML which can be used to interactively simulate a communicating process. The input of the interpreter is a stream of oracle trees determining which process talks at each meeting point, and its output is the resulting discourse of the process. It can be run interactively using a lazy functional language, so that each time the user enters an oracle tree, he/she learns which message is broadcasted. Before building the proof, we used this interpreter to gain a better understanding of the protocol. As a by-side product of the expertiment, we also validated it in Coq using the Program family of tactics [14, 15]. These tactics provide assistance to the user in the task of verifying already-written programs with respect to certain initial specification of them. The proof consisted in showing that the stream of messages yielded by the interpreter is actually a discourse, i.e., that it actually comes from a trace of the process. This proof is also available within the contributions of the version V5.10 of Coq.

<sup>&</sup>lt;sup>5</sup> This is not strictly necessary, but it simplifies the proof.

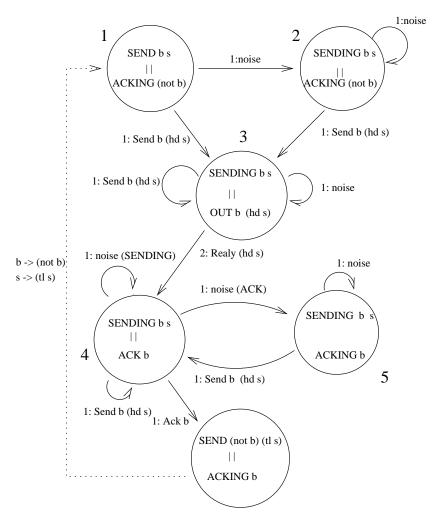


Figure 3: The states of the protocol.

#### 4.3 The formal proof

The intuition about the correctness of the protocol arises from the transition diagram in Figure 3, which describes the possible states in the communication process. Let us analyse this cyclic diagram starting from State 1. The sender wants to send a value and the receiver is ready to hear it. The sender may fail its transmission going into State 2, but it keeps on repeating the value sent until it is acknowledge. Thus, if it does a clear transmission later on, it will recover from this failure and go into State 3. At State 3 the receiver has heard the value and the synchronisation bit is the expected one, so it changes its own bit and tries to repeat the message heard on channel two. We realise at this point that the way in which the two processes are scheduled actually matters: if the sender speaks all the time and the receiver never does it, the system will be blocked. If the receiver succeeds in talking on channel two, the process goes into the acknowledgment phase. Again, if the sender lets the receiver talk, it will send the expected acknowledgment, and the system reaches (an instance of) State 1 again. If at State 4 the transmission of the receiver fails, the receiver hangs waiting for a new message from the sender, and it will realise about the failure comparing its synchronisation bit with the one re-transmitted by the sender. At this moment it comes back to State 3 and tries to re-send the acknowledgment to the sender. The process breaks through from the acknowledgment phase only when the synchronisation bit of the receiver has been clearly heard by the sender.

The diagram shows that there are two requirements which are necessary for ensuring the correctness of

the protocol:

- 1. Channel one is fair, i.e. each of the process is capable of sending a value clearly after a finite number of attempts. This avoids the system to be blocked looping in states 2, 3, 4 and 5, or between states 4 and 5.
- 2. When the receiver wants to talk, it is capable of doing it. In other words, both processes are scheduled equitably. This avoids infinite loops in states 3 and 4.

These constraints on the possible evolutions of the system can be expressed using the general property of cyclic determination described in the previous section, where the property determined to fail is the disjunction of the the possibility for the signal performed to be a  $\tau$  and property Left TalksSafely defined in example 3.2. This amounts to say that we are excluding those traces containing infinite communications made of failed and/or sender's transmitions. The constraint are only for channel one, so if the action is on channel two we consider that the property has faild. Thus, we define the restriction on the traces as the following predicate Restriction:

```
 \begin{aligned} & \text{Definition } & TransConditions := \\ & [c:Channel][w:(Signal\ Act)][p,q:Process] \\ & \text{Case } c \text{ of } \\ & [t:(UnrelProcTrans\ p\ !w\ q)]w = \tau \lor (LeftTalksSafely\ chnl_1\ p\ !w\ q\ t) \\ & [t:(SafeProcTrans\ p\ !s\ q)] \ True \\ & \text{end} \\ & \text{Definition } & Restriction := (CycDet\ Trans\ TransConditions). \end{aligned}
```

Remark that the restriction of fairness mentioned in the first requirement above is hard to express in this framework just as a property of the discourse of the system, because it is in fact a property of the stream of communications. The naive constraint saying that the discourse does not contain an infinite sub-stream of Noise signals is not enough, since at State 4 the discourse Noise, (Send b (hd s)), Noise, (Send b (hd s))... verifies this property, but it anyway corresponds to a beheavior of the system in which the receiver always looses its transmissions. It is therefore necessary to also take care of who fails in sending a message, in order to verify that it is not always the same process. This can be determined only analysing the transition proof, which shows who talked in the communication.

The property to be verified is that any discourse verifying the restriction above contains (in order) the stream of messages  $(Relay s_0), (Relay s_1), \ldots$  on channel two, where  $s_0, s_1, \ldots$  is the stream of input messages of the sender. This property can be expressed as an instance of the property SerDet introduced in section 3.3, where the general parameter B corresponds to the type of messages A, and the i-th property to fail is the negation that the message  $(Relay s_i)$  has been heard on channel two.

```
 \begin{aligned} \text{Definition } MessageLost := \\ & [a:A][c:Channel][w:Signal][p,q:Process] \\ & [t:(UnrelProcTrans\ c\ p\ w\ q)] \neg (c = chnl_2 \land w = (Clear\ (Relay\ b))\ ). \end{aligned} \\ \text{Definition } InTrace := (SerDet\ Trans\ A\ MessageLost). \\ \text{So the theorem to be proved is:} \\ \text{Theorem } ABP\_Correctness: \\ & \forall b:bool.\ \forall s:(Stream\ A).\ \forall ss:(Stream\ Signal). \\ & (Restriction\ (ABP\ b\ s)\ ss) \Rightarrow (InTrace\ s\ (ABP\ b\ s)\ ss). \end{aligned}
```

This theorem follows immediately from the following lemma and the fact that ABP re-transmits the head of the stream twice.

```
Lemma cycle:

\forall b : bool. \ \forall s : (Stream \ A). \ \forall ss : (Stream \ Signal).

(Restriction \ (PAR \ (SENDING \ b \ s) \ (ACK \ b)) \ ss) \Rightarrow

(In Trace \ (tl \ s) \ (PAR \ (SENDING \ b \ s) \ (ACK \ b)) \ ss).
```

The method used in the proof of cycle combines guarded definitions and a form of reasoning which is quite reminiscent of Brouwer's principle of bar induction [5]. Let us consider again that all the possible traces from (ABP b s) verifying the restriction are arranged in a tree. Such a tree can be associated to what Brouwer called a spread, and the property Restriction to the corresponding spread-law. The steps of our proof consists in showing the following facts:

- 1. Hearing the first message of the stream is a property determined to happen. In terms of the principle of bar induction, this corresponds to the requirement of showing that there is a property which bars the spread.
- 2. Any witness q of this fact is an instance (ABP (not b) (tls)) of the process at the top of the tree.<sup>6</sup>. This property is proved by induction on the proof that q is a witness. This induction amounts to prove that it satisfies the other two premises required in the bar induction principle:
  - it follows from the one barring the spread (base case);
  - it is "hereditary upwards" through the branches of the tree (inductive case).
- 3. The restriction still holds for the stream of signals starting from the (unique) witness of the fact that the first message has been repeated on channel two. This fact enables us to grasp the (infinite) proof of the lemma as a cyclic repetition of the finite one carried out upon here.

The infinite repetition of barrings of the spread in which the proof consists corresponds to an infinite application of the only constructor of the property SerDet, which is described through a guarded definition using the command CoFixpoint The proof does not make use of any explicit axiom of bar induction, but this principle is implicit in the assumption that the Steps 1 and 2 above ensure that all the traces from State 3 contain this state further on.

#### 4.3.1 Step one: the tree is barred

Assume a boolean b, a stream of messages s, a process p and a stream of signals ss. The first step of the proof is expressed by the following theorem.

```
Theorem Bar_{4,5}:
(Restriction \ p \ ss)
\Rightarrow p = (PAR \ (SENDING \ b \ s) \ (ACK \ b)) \lor p = (PAR \ (SENDING \ b \ s) \ (ACKING \ b))
\Rightarrow (Det \ Trans \ (MessageLost \ (hd \ (tl \ s))) \ p \ ss).
```

As was already said, the first message of the stream will not be taken into consideration by the receiver, but loosing the second message (the message at the head of the tail of messages) is a property determined to fail. The proof is by induction on the proof of (simple) determination contained in the restriction, using three other similar lemmas for the previous sates:

<sup>&</sup>lt;sup>6</sup>Is in order to obtain this property that we have chosen to start the protocol at State 4. However, note that this choice obliges us to add a dummy message at the beginning of the stream, because at the first transmition the sender will be waiting for the acknowledgment of a "previous" (inexistent!) message to start repeating the next one (see Figure 3).

The proofs of all these lemmas are very similar. The general structure consists in analysing which are the possible states the process could run into, and using the local restriction to show that it will break through those situations that could block it. The statement of theorem  $Bar_{4,5}$  is a bit different from those of the lemmas. States 4 or 5 have to be considered at once, and the proof needs the fact that the restriction is not only simple but cyclicly determined to happen. The simple determination of the restriction is used to prove that, after some looping between states 4 and 5, the receiver finally sends its acknowledgment clearly. However, it is necessary to show that the restriction still holds after this, so that the hypothesis to apply the lemma for state 1 can be fullfiled. In other words, if the process is in the acknowledgment phase, the receiver has to speak clearly twice to send the message on channel two: one to break through this phase and then once more on channel two.

Studying the possible situations into which a process p could evolve amounts to derive from a proof of  $p \xrightarrow{\circ}_w q$  which are the possible instances for c, w and q. Fortunately, this problem (which is terribly tedious when doing manually) can be automatically solved in the V5.10 of Coq using the so-called *inversion tactics* [4]. The help provided by these tactics is quite remarkable, and save us several hours of dealing with boring proofs<sup>7</sup>. So the proofs of these lemmas are quite long (about 60 Coq tactics each one) but they present no difficulties.

The fact that most of these proofs consist in inverting the transition predicate explains why also the alternative of using boxes proposed in section 3 for the representation of processes had to be discharged. If we would consider the language spoken as an index of the type of processes, so we should have done in the transition predicates, and in this case inversion tactics would not work. This is due to the fact that these tactics can not deal with the inversion of predicates where the type of some of its indexes depend on previous indexes of the predicate (cf. [4]).

#### 4.3.2 Step two: there is a unique witness

Assumme a boolean b, a stream of messages s, two process p,q and two streams of signals ss,rss. The second requirement is expressed by this theorem :

```
Theorem who_is_the_witness<sub>4,5</sub>: (Witness Trans (MessageLost (hd (tls))) q rss p ss) \Rightarrow (p = (PAR (SENDING b s) (ACK b)) \lor p = (PAR (SENDING b s) (ACKING b))) \Rightarrow q = (PAR (SENDING (not b) (tls)) (ACK (not b))).
```

stating which is the only possible witness of the fact that the message has been correctly received. The proof is by induction on the proof that q is a witness, analysing the possible forms this witness could take if we start from p. It also need similar lemmas for the previous states.

```
Lemma who_is_the_witness_3: (Witness\ Trans\ (MessageLost\ (hd\ s))\ q\ rss\ p\ ss) \Rightarrow p = (PAR\ (SENDING\ b\ s)\ (OUT\ b\ (hd\ s))) \Rightarrow q = (PAR\ (SENDING\ b\ s)\ (ACK\ b)). Lemma who_is_the_witness_2: (Witness\ Trans\ (MessageLost\ (hd\ s))\ q\ rss\ p\ ss) \Rightarrow p = (PAR\ (SENDING\ b\ s)\ (ACKING\ (not\ b))) \Rightarrow q = (PAR\ (SENDING\ b\ s)\ (ACK\ b)). Lemma who_is_the_witness_1: (Witness\ Trans\ (MessageLost\ (hd\ s))\ q\ rss\ p\ ss) \Rightarrow p = (PAR\ (SEND\ b\ s)\ (ACKING\ (not\ b))) \Rightarrow q = (PAR\ (SEND\ b\ s)\ (ACKING\ (not\ b))).
```

 $<sup>^{7}</sup>$ Just to give an idea, a single one of the 19 inversions necessary to prove  $Bar_{1}$  represents the application of about 100 basic tactics.

#### 4.3.3 Step three: and so on forever

In order to close the description of the infinite proof with a recursive call, it must be shown that the restriction still holds for the stream of signals remaining from the (unique) witness at which the first message is repeated on channel two. This property follows from the lemma StillHappens referred in section 3.3. This lemma requires MessageLost to be decidable, and this follows straightforwardly form the decidability of LeftTalksSafely. This latter property is immediate when LeftTalksSafely is defined as a propositional function using case analysis, but a little more harder if it is introduced as an inductive predicate.

#### 5 Concluding Remarks

We have illustrated the use of co-inductive types in a prototype version of Coq through a case of study. In our opinion, the best advantage of this prototype is to combine a simple and direct way of describing co-inductive types and their elements with a safe way of reasoning about them. Other proofs systems which allow to reason about infinite objects are Alf [13] and Isabelle [19]. The main differences with the former is that Alf does not prevent the user from the introduction of non-sense definitions, since the verification of the guarded condition is up to him. Another difference is that in Alf type conversion may diverge when working with infinite objects. This is because non-ending methods of construction are not considered as canonic expressions, so that testing the equality of two infinite objects may lead to an infinite unfolding of their definitions. In Isabelle, co-inductive types can be encoded as greatest fixed points of monotone operators, and the way of introducing infinite objects is through co-recursive operators [18, 20]. This encoding gives exactly the same strength than using guarded definitions [8]. There exists packages for working with (co)inductive types, which automatise part of this encoding and derives a co-recursive operator associated with the type. However, infinite objects have to be encoded by hand in terms of this co-recursor, which is a rather tedious task in practice, because it makes the codification of the type explicit again.

A verification proof of the alternating bit protocol in Coq has been previously developed by M. Bezem and J.F. Groote [2]. The fact that their proof is based on a completely different formalism (the process algebra  $\mu$ CRL [25]) makes the comparison with our's difficult. We share with [2] the use of a (co)algebraic language for describing process, but the way of giving semantics to the language is quite different. While in  $\mu$ CRL an axiomatic approach is followed, stating properties about the equality of the process algebra, in CBS the style is closer to natural semantics, where the meaning is given using a transition relation. It is remarkable that the the size of our proof is almost a third of the proof in [2] (about 70 Kb vs. 200 Kb of Coq's source<sup>8</sup>). In our opinion, one of the facts that could explain this difference of size is that in [2] the equality of the algebra is represented in Coq as Leibniz's propositional equality. This choice leads to a proof requiring a lot of rewriting, and unfortunately Coq still lacks of powerful rewriting tactics. On the contrary, the use of a transition relation specified as an recursive predicate allowed us to take advantage of Coq's specialised tactics for reasoning about recursive types, for example to deduce constraints about the subprocesses of a given one by inversion (cf. 4.3).

Other process models that have been experimented in Coq are based on I/O automata, see for example [9]. The approach of CBS is close to these models, but it is more syntactical in nature. In CBS the emphasis is put on the language of processes, and the transition relation is not an arbitrary one as in I/O automata but follows the structure of this language, as in the natural semantics style. An advantage of this fact is that, describing this language as a co-inductive type, the process specified can be immediately translated into real programs written in a (lazy) functional programming language. In this way, the verification of the protocol also gives a certified prototype that can be used to experiment with it. This fits well with Coq's doctrine of program extraction as a way of developping certified software [17]. However, Coq's mechanism of program extraction has to be revisited in order to take full advantage of this adequacy. The present way of specifying the computational parts of an indexed type is not as powerful as needed. In particular, it is annoying that when extracting the computational contents of an indexed type it is not possible to forget the indices in their constructors. This prevents, for example, the possibility of considering a discourse as the computational contents of a trace. Furthermore, it should be interesting to be able to obtain different extracted programs

 $<sup>^8</sup>$ Bezem's original proof has been developed in the version V5.8 of Coq. This comparison has been done on the proof ported by the Coq team on the version V5.10 of the system

depending on which arguments in the constructors are considered as having computational contents. These limitation in the extraction mechanism based on marking has been already mentioned in the experiment described in [12], and they seem to appear quite often when working with co-inductive types.

Another way of modeling communicating processes using co-inductive types is through Kahn's stream networks [10]. In this model the sender and the receiver are described as the solution of an array of mutually dependent equations defining the streams of messages sent by each of the communicating agents. Several proofs of this protocol have been based on this model, see for example [26, 6, 11]. Unfortunately, most of the time the equations of these systems, even though representing admissible methods of constructing a stream, are not guarded by constructors, and thus they can not be expressed by a CoFixpoint definition. It also seems to be easy to extract programs for simulating the process from a specification given in terms of Kahn's networks.

The complete development of the proof took us about a month of work and yields 70 Kb of source code. At least half of the time was invested in looking for good axiomatisations of CBS in Coq. This experiment gives us also the opportunity for testing some of the new features of the version V5.10 of Coq. We have already mentioned the important help provided by inversion tactics and proof reconstruction facilities. Among the main technical difficulties detected is the absence of simple mechanisms to handle the theories defined, for example simple ways of instantiating, hiding and inferring their general parameters. Also, some of the new features of Coq –like the possibility of extending the grammar and the pretty printer– still lacks of some form of assistance from the system to take full advantage of them.

The carried proof out illustrates the application of some proof techniques that can be used when reasoning about infinite objects, like the principle of bar induction and the construction of non-ending proofs using guarded definitions. To our knowledge, there are not many examples of the applications of these principles to Computer Science problems. Their integration into Coq's proof editor has to be better studied and is part of the future work. Some of the problems for this integration seems to come from the fact that the guard condition is considered as a side property imposed on the rough terms of the calculus [8]. This has lead us to study an alternative formulation of the extension presented in that paper, when the guarded condition is integrated as part of the typing judgments of the theory. We hope that this could help in the implementation, and also provide a simpler presentation of the theory. Such a simplification also seems to be necessary in order to well establish the meta-theoretical properties of the extension underlying the prototype.

#### References

- [1] Peter Aczel. Non-Well-Founded Sets, volume 14 of CLSI Lecture Notes. 1988.
- [2] M. Bezem and J. Groote. A Formal verification of Alternating Protocol in the Calculus of Constructions. Technical Report 88, Departament of Philosophy, University of Utrecht, 1993.
- [3] T. Coquand. Infinite objects in Type Theory. In Henk Barendregt, Tobias Nipkow, editor, Types for Proofs and Programs, pages 62–78. LNCS 806, 1993.
- [4] C. Cornes and D. Terrasse. Inverting predicates in Coq. In BRA Workshop on Types for Proofs on Programs, 1995.
- [5] M. Dummett. Elements of Intuitionism. Oxford University Press, 1977.
- [6] P. Dybjer and H.P. Sanders. A functional programming approach to the specification and verification of concurrent systems. Formal Aspects of Computing, 1:303-318, 1989.
- [7] C. Cornes et al. The Coq Proof assistant user's guide Version V5.10. Technical Report 0177, INRIA, 1995.
- [8] E. Giménez. Codifying guarded definitions with recursive schemes. In *BRA Workshop on Types for Proofs and Programs*, 1994. To appear in the LNCS series. An extended version of the paper is also available as Research Report 95-07 of LIP, Ecole Normale Supérieure de Lyon, at ftp.ens-lyon.fr/pub/Rapports/RR/RR95/RR95-07.ps.Z

- [9] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-Checking a Data Link Protocol. In Henk Barendregt, Tobias Nipkow, editor, *Types for Proofs and Programs*, pages 127–165. LNCS 806, 1993.
- [10] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. *Information Processing* 77, pages 993–998, 1977.
- [11] F. Leclerc. Etude de la programmation de systèmes parallèles par flot de données dans les langages fonctionnels typés. Master's thesis, Ecole Normale Supérieure de Lyon, 1992.
- [12] F. Leclerc and C. Paulin-Mohring. Programming with streams in Coq. A case study: the Sieve of Eratosthenes. In Henk Barendregt, Tobias Nipkow, editor, Types for Proofs and Programs, pages 191– 212. LNCS 806, 1993.
- [13] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In Nijmegen Workshop on Types for Proofs and Programs, 1993.
- [14] C. Parent. Developping certified programs in the system Coq: the Program tactic. In Henk Barendregt, Tobias Nipkow, editor, Types for Proofs and Programs, LNCS 806, pages 291-312, 1993.
- [15] C. Parent. Synthesizing proofs from programs in the Calculus of Inductive Constructions. In Mathematics for Programs Constructions '95, LNCS 947, pages 351-379, 1995.
- [16] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and Properties. In M. Bezem, J.F. Groote, editor, *Proceedings of the TLCA*, 1993.
- [17] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. Journal of Symbolic Computation, 15:607-640, 1993.
- [18] L. Paulson. Co-induction and Co-recursion in Higher-order Logic. Technical Report 304, Computer Laboratory, University of Cambridge, 1993.
- [19] L. Paulson. The Isabelle reference manual. Technical Report 283, Computer Laboratory, University of Cambridge, 1993.
- [20] L. Paulson. A fixed point approach to implementing (co)inductive definitions. Technical Report 304, Computer Laboratory, University of Cambridge, 1995.
- [21] K.V. Prasad. A calculus of broadcasting systems. In S. Abramsky, T. Maibaum, editor, *TAPSOFT'91*, pages 338–358. LNCS 493, 1991.
- [22] K.V. Prasad. Programming with broadcasts. In E. Best, editor, CONCUR'93, pages 173–187. LNCS 715, 1993.
- [23] K.V. Prasad. A Calculus of Broadcasting Systems. To appear in *Science of Computer Programming*, 1995.
- [24] H.P. Sanders. A Logic of Functional Programs with an Application to Concurrency. PhD thesis, Chalmers University of Göteborg, 1992.
- [25] M. P. A. Sellink. Verifying process algebra proofs in type theory. Technical report, Departament of Philosophy, University of Utrecht, 1993.
- [26] T. Streicher. A verification method for finite dateflow networks with constraints applied to the verification of the alternating bit protocol. Technical Report MIP-8706, Passau University, 1987.