Formal Description and Analysis of a Bounded Retransmission Protocol

Radu Mateescu

N° 2965

August 1996

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--2965--FR+ENG

Formal Description and Analysis of a Bounded Retransmission Protocol

Radu Mateescu[*]

Thème 1 — Réseaux et systèmes
Projet Spectre

Rapport de recherche n° 2965 — August 1996 — 28 pages

Abstract: This paper reports about the formal specification and verification of a Bounded Retransmission Protocol (BRP) used by Philips in one of its products.

We started with the descriptions of the BRP service (i.e., external behaviour) and protocol written in the μCRL language by Groote and van de Pol. After translating them in the LOTOS language, we performed verifications by model-checking using the CADP (CÆSAR/ALDÉBARAN) toolbox.

The models of the LOTOS descriptions were generated using the CÆSAR compiler (by putting bounds on the data domains) and checked to be branching equivalent using the ALDÉBARAN tool.

Alternately, we formulated in the ACTL temporal logic a set of safety and liveness properties for the BRP protocol and checked them on the corresponding model using our XTL generic model-checker.

Key-words: Formal methods, Formal description techniques, Communication protocols, Protocol engineering, LOTOS, Verification, Validation, Model-checking, Labelled Transition Systems, Bisimulation, Temporal logic.

This report is also available as “Formal Description and Analysis of a Bounded Retransmission Protocol”, in Z. Brezočnik and T. Kapus, editors, *Proceedings of COST 247 International Workshop on Applied Formal Methods in System Design (Maribor, Slovenia)*, University of Maribor, Slovenia, June 1996.

[*] E-mail: `Radu.Mateescu@imag.fr`

Description formelle et analyse d'un protocole à retransmission bornée

Résumé : Ce rapport présente la spécification formelle et la vérification du protocole Brp (Bounded Retransmission Protocol) utilisé par Philips dans l'un de ses produits.

Nous nous sommes basés sur les descriptions du service (comportement externe) et du protocole Brp écrites en μCrl par Groote et van de Pol. Après les avoir traduites en Lotos, nous avons effectué des vérifications basées sur les modèles (*model-checking*) à l'aide de la boîte à outils Cadp (Cæsar/Aldébaran).

Nous avons engendré les modèles correspondant à ces descriptions Lotos à l'aide du compilateur Cæsar (en imposant des bornes aux domaines des données manipulées) et nous avons vérifié, à l'aide de l'outil Aldébaran, que ces modèles sont équivalents modulo la bisimulation de branchement.

Par ailleurs, nous avons aussi formulé en logique temporelle Actl un ensemble de propriétés de sûreté et de vivacité du protocole Brp et nous les avons vérifiées sur le modèle correspondant à l'aide de notre évaluateur générique Xtl.

Mots-clés : Méthodes formelles, Techniques de description formelle, Protocoles de communication, Ingénierie des protocoles, Lotos, Vérification, Validation, Systèmes de transitions étiquetées, Bisimulation, Logique temporelle.

1 Introduction

The necessity of formal verification in the designing of distributed systems and communication protocols is now widely recognized. Over the last years, a wide range of formalisms and methodologies dealing with the description of concurrent applications and the specification of their expected properties have been defined.

There are essentially two approaches to formal verification, namely *theorem-proving* and *model-checking*. Both have been extensively studied, and various algorithms and tools have been developed. Theorem-proving allows to deal with infinite-state systems, but it cannot be fully automated. On the other hand, model-checking techniques, although restricted to finite-state systems, provide a simple and efficient approach, especially useful in the early steps of the design process, where the errors are likely to occur more frequently.

This paper deals with the formal description and verification by model-checking of the Bounded Retransmission Protocol (BRP) used by Philips in one of its products.

This protocol has already been studied using different approaches. Groote and van de Pol [GvdP93] used the μCRL specification language [GP90] to describe the respective behaviours of the BRP service (i.e., external behaviour) and protocol, proved that they are branching equivalent, and then computer-checked the proof using the COQ system [DFH+93]. Helmink, Sellink and Vaandrager [HSV94] modeled the BRP protocol and service in the framework of I/O automata theory [LT89], proved its correctness and (partially) computer-checked the proof using the COQ system. Havelund and Shankar [HS96] abstracted a finite-state version of the BRP protocol using the PVS theorem prover [ORS92] and then verified its correctness by model-checking with the MURΦ state exploration tool [MDI92].

In our approach we selected the ISO language LOTOS [ISO88b] to describe the BRP protocol and service. We started from the μCRL descriptions written by Groote and van de Pol, which we translated into LOTOS. Since both μCRL and LOTOS languages are based on abstract data types and process algebras, this translation was quite straightforward.

Regarding verification of the correctness, we used model-checking rather than theorem-proving. The verification was performed using the CADP (CÆSAR/ALDÉBARAN) toolbox [FGK+96] by means of two different methods: bisimulations and temporal logics.

The paper is organized as follows. Section 2 presents briefly the LOTOS language. Section 3 describes the CADP protocol engineering toolbox. Section 4 gives the LOTOS descriptions of the BRP service and protocol. Section 5 presents the generation of the models corresponding to the two LOTOS descriptions. Sections 6 and 7 describe the various verifications performed by means of bisimulations and temporal logics. Finally, Section 8 gives some concluding remarks and Annex A contains the LOTOS description of the data structures used in the protocol.

2 The ISO language LOTOS

Many formalisms have been proposed for describing parallel systems, among which the standardized Formal Description Technique LOTOS[1] has received a considerable attention from the research community.

[1]Language Of Temporal Ordering Specification

LOTOS is a formal language intended for the specification of communication protocols and distributed systems. It was developed during the years 1981–88 in the framework of the SEDOS[2] project and standardized by ISO[3] in 1988 [ISO88b]. Several tutorials for LOTOS are available, e.g. [BB88, Tur93].

The design of LOTOS was motivated by the need for a language with a high abstraction level and a strong mathematical basis, which could be used for the description and analysis of complex systems. As a design choice, LOTOS consists of two "orthogonal" sub-languages:

The data part of LOTOS is dedicated to the description of data structures. It is based on the well-known theory of algebraic abstract data types [Gut77], more specifically on the ACTONE specification language [EM85, dMRV92].

In this approach, data structures are described by *sorts*, which represent value domains, and *operations*, which are mathematical functions defined on these domains. The meaning of operations is defined by algebraic *equations*. Sorts, operations, and equations are grouped in modules called *types*, which can be combined together using importation, renaming, parametrization, and actualization. The underlying semantics is based on initial algebras [EM85].

The control part of LOTOS is based on the process algebra approach for concurrency, and appears to combine the best features of CCS [Mil89] and CSP [Hoa85]. As a process algebra, LOTOS relies on a small set of basic operators, which represent primitive concepts of concurrent systems (sequential composition, non-deterministic choice, guard, parallel composition, rendez-vous, etc.). These operators are used to build algebraic terms in a compositional way, since complex behaviours can be obtained by combining elementary ones.

As for most process algebras, the semantics of LOTOS is formally defined in terms of *labeled transition systems* (LTSs for short) [Par81, Mil89, ISO88b], i.e., directed graphs whose vertices denote the global *states* of the system and whose arcs correspond to the evolutions (*transitions*) of the system.

LOTOS has been applied to describe complex systems formally, for example: the service and protocols for the OSI transport and session layers [ISO89b, ISO89a, ISO92c, ISO92d], the CCR[4] service and protocol [ISO95b, ISO95a], OSI TP[5] [ISO92b, Annex H], MAA[6] [ISO92a, Mun91], FTAM[7] basic file protocol [ISO88a, LL95], etc. It has been mostly used to describe software systems, although there are several attempts to use it for asynchronous hardware description [CGM+96].

A number of tools have been developed for LOTOS, which cover most user needs in the areas of simulation, compilation, test generation and formal verification. For this case-study, we used the CADP toolbox [FGK+96], which provides state-of-the-art verification features.

Since 1993, a revision of the LOTOS standard has been undertaken within ISO. This on-going activity should give birth to a new version of LOTOS, named E-LOTOS (for *Extended LOTOS*), which is expected to introduce simpler features (especially for data types) and increased expressiveness (notably by adding quantitative time to the language). However, this new language being not available at the time of this case-study, we based our work on the existing standard LOTOS.

[2] Software Environment for the Design of Open Distributed Systems, ESPRIT project 410
[3] International Organization for Standardization
[4] Commitment, Concurrency, and Recovery
[5] Distributed Transaction Processing
[6] Message Authentication Algorithm
[7] File Transfer, Access and Management

3 The CADP verification toolbox

The CADP[8] toolbox is dedicated to the design and verification of communication protocols and distributed systems. Initiated in 1986, several motivations have contributed to its development since this date:

- This toolbox aims to offer an integrated set of functionalities ranging from interactive simulation to exhaustive, model-based verification methods. In particular, both logical and behavioural specifications can be verified.
- One of the major objectives of the toolbox is to deal with large case studies. Therefore, in addition to enumerative verification methods, it also includes more sophisticated approaches, such as symbolic verification, on-the-fly verification, and compositional model generation.
- Finally, this toolbox can be viewed as an open software platform: in addition to LOTOS, it also supports lower-level formalisms such as finite state machines and networks of communicating automata.

In the sequel we only present the tools used throughout this case-study:

- CÆSAR [GS90] and CÆSAR.ADT [Gar89, GT93] are compilers that translate a LOTOS program into an LTS describing its exhaustive behaviour. This LTS can be represented either *explicitly*, as a set of states and transitions, or *implicitly*, as a library of C functions allowing to execute the program behaviour in a controlled way.
- ALDÉBARAN [Fer90, FKM93] is a verification tool able to compare or to minimize LTSs with respect to (bi)simulation relations [Par81, Mil89]. Initially designed to deal with explicit LTSs produced by CÆSAR, it has been extended to also handle networks of communicating automata (for on-the-fly and symbolic verification).

  Several simulation and bisimulation relations are implemented within ALDÉBARAN, which offers a wide spectrum for expressing such behavioural specifications.
- XTL (eXecutable Temporal Language) [Mat94] is a functional-like programming language designed to allow an easy, compact implementation of various temporal logic operators that are evaluated over an LTS generated from a source program.

  Beside the usual predefined types (booleans, integers, characters, strings, etc.), the XTL language allows to access all types and functions defined in the source program and provides special types, such as states, transitions, sets of states, sets of transitions, and labels of the LTS. It offers primitives to access the informations contained in states and labels: this allows to express "basic predicates" (i.e., containing no temporal operators) defined over the states and labels of the LTS. There also exist predefined functions to access the initial state and the successors and predecessors of states and transitions, thus allowing the exploring of the transition relation.

  The temporal operators of various temporal logics can be implemented as recursively defined user XTL functions operating on sets of states and transitions. A prototype compiler for XTL has been developed, and several temporal logics like HML [HM85], CTL [CES86], LTAC [QS83] or ACTL [NV90] have already been implemented in XTL.

[8] CÆSAR/ALDÉBARAN Development Package

4 Lotos description of the Brp

The Brp protocol is designed to transmit (large) data packets over an unreliable medium by splitting them in (small) chunks, which are sent sequentially. After each chunk transmission, the *sender entity* of the protocol waits for an acknowledgement from the *receiver entity* before sending the next chunk. In case a transmission failure occurs and the acknowledgement fails to come, the sender times out and retransmits the chunk. Only a limited number of retransmissions is allowed (*bounded retransmission*); if this limit is reached, the protocol aborts the transmission of the data packet, appropriately informing the *sending client* and the *receiving client*.

This kind of protocol is known as "positive acknowledgement with retransmission" protocol [MV93] since, unlike the Alternating Bit Protocol, the acknowledgements sent by the receiver carry no information.

4.1 The Brp service

We present below a Lotos description of the Brp service, i.e., the external behaviour that the protocol should have while interacting with a sending client and a receiving client.

The Brp service is modeled by the Lotos process `SERVICE` below. It communicates with the sending and receiving clients via the gates `INPUT` and `OUTPUT`, respectively.

The data packets produced by the sending client, modeled as lists of data chunks (see Annex A for a description of the type `Packet`), are read at the `INPUT` gate. After accepting a packet, a decision is made either to reject the packet if it is empty[9] and to go back to the initial state (modeled by a recursive call of the `SERVICE` process) or to accept it and begin its transmission (modeled by a call to the `SERVICE_1` process).

```
process SERVICE [INPUT, OUTPUT] : noexit :=
   INPUT ?P:Packet;
      (
      [len (P) == 0] -> (* empty packets rejected *)
         SERVICE [INPUT, OUTPUT]
      []
      [len (P) > 0] ->
         SERVICE_1 [INPUT, OUTPUT] (P, len (P))
      )
endproc
```

The `SERVICE_1` process has two value parameters: `P`, which represents the portion of the initial packet that remains to be transmitted, and `L`, which stands for the length (i.e., number of chunks) of the initial packet.

In case of a faultless transmission, each chunk is delivered to the receiving client at the `OUTPUT` gate, accompanied by an `I_FST`, `I_INC` or `I_OK` indication. `I_FST` is used for the first (but not the last) chunk of a packet, `I_INC` stands for an intermediate (but not last) chunk, and `I_OK` accompanies the last chunk of the packet. If the whole packet has been successfully transmitted, an `I_OK` confirmation is issued at the `INPUT` gate and the protocol is ready to read a new data packet. The indications are modeled as values of an enumerated type `INDICATION` (see Annex A). Two auxiliary functions `ind` and `conf` are used to calculate the indication for the receiving client and the confirmation to the

[9]We follow here the description given by Helmink and al. where the empty data packets are not accepted by the protocol. In Groote and van de Pol's description, a special value *head (empty)* is sent in this case to the receiver.

sending client. The `head` and `tail` operators are used to select the first chunk *resp.* the remainder of a packet, and the `len` operator computes the number of chunks in a packet.

In case of transmission failure, the sending client is informed via an `I_NOK` ("not OK") confirmation at the `INPUT` gate. If the failure occurs in the middle of a packet (i.e., after delivery of the first chunk but before the last chunk is delivered), the receiving client is informed also by an `I_NOK` indication delivered without data at the `OUTPUT` gate.

However, if the last acknowledgement is lost, an `I_DK` ("don't know") confirmation is issued at the `INPUT` gate, because in this case there is no way the sending client can know whether the last chunk has been delivered or not. After an `I_DK` confirmation, the protocol is ready to accept a new data packet.

The `i` action represents the *silent* action (noted τ in CCS) and is used here to prevent the sending client from forcing a successful result of a transmission. For example, the choice between the rendezvous "`INPUT !I_OK`" and "`INPUT !I_DK`" below cannot be influenced by the sending client, since the protocol can always perform non-deterministically one of the silent actions `i`.

```
process SERVICE_1 [INPUT, OUTPUT] (P:Packet, L:Nat) : noexit :=
   i;
      OUTPUT !head (P) !ind (len (P) == L, len (P) == 1);
         (
         [len (P) == 1] -> (* last chunk delivered *)
            (
            i;
               INPUT !I_OK;
                  SERVICE [INPUT, OUTPUT]
            []
            i;
               INPUT !I_DK;
                  SERVICE [INPUT, OUTPUT]
            )
         []
         [len (P) > 1] ->  (* more chunks to deliver *)
            (
            i;
               SERVICE_1 [INPUT, OUTPUT] (tail (P), L)
            []
            i;
               INPUT !I_NOK;
                  OUTPUT !I_NOK;
                     SERVICE [INPUT, OUTPUT]
            )
         )
   []
   i;
      (
      [len (P) == L] ->    (* failure at the first chunk *)
         INPUT !conf (P);
            SERVICE [INPUT, OUTPUT]
      []
      [len (P) < L] ->     (* failure at an intermediate chunk *)
```

```
             INPUT !conf (P);
                OUTPUT !I_NOK;
                   SERVICE [INPUT, OUTPUT]
          )
    endproc
```

4.2 The Brp protocol

4.2.1 The protocol architecture

Like most communication protocols, the Brp protocol can be described by means of a sender and a receiver entities exchanging messages via two channels. However, the actual architecture of the protocol is slightly more complicated due to the need for timers. The architecture of the protocol modeled in Lotos is shown on Figure 1.

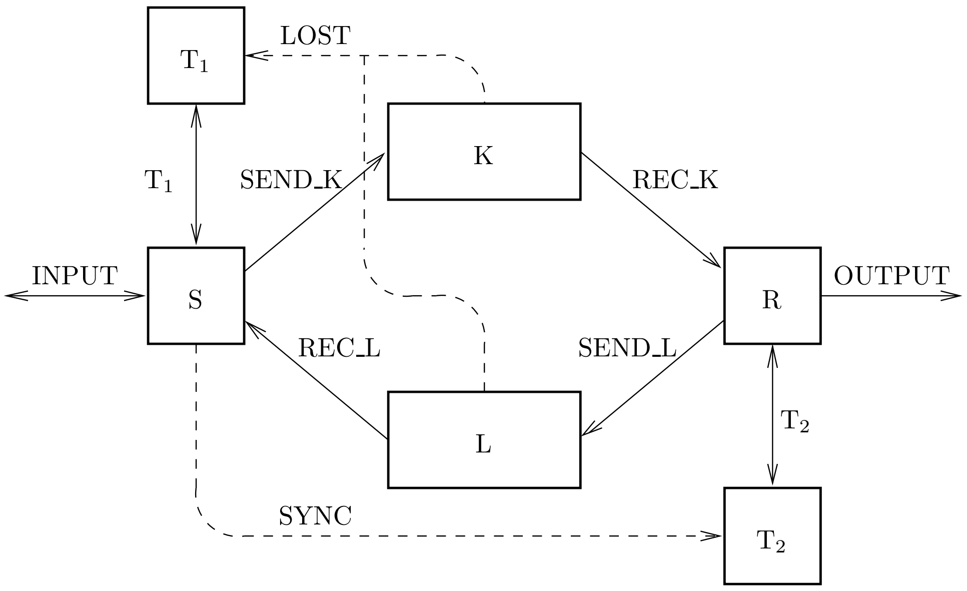

Figure 1: The architecture of the Brp protocol

The sender `S` communicates with the sending client via the `INPUT` gate, through which it receives the data packets and sends the confirmations back. The chunks are sent to the channel `K` at the `SEND_K` gate and the acknowledgements are received from the channel `L` at the `REC_L` gate. The sender is equipped with a timer `T1` that it can start and reset by sending signals at the `T1` gate. The timer can send a timeout signal via the same gate.

The receiver `R` receives the data chunks from the channel `K` via the `REC_K` gate, delivers them at the receiving client via the `OUTPUT` gate and sends the acknowledgements to channel `L` at the `SEND_L` gate. It is also equipped with a timer `T2`, which can be started and reset by signals at the `T2` gate and can issue a timeout signal at the same gate.

Since time cannot be explicitly manipulated in Lotos, we use extra synchronization links (represented by dashed lines on Figure 1) to ensure that causality constraints are not violated. For example, we must make sure that timer `T1`, which is started when a chunk is sent, times out only if the corresponding acknowledgement fails to come (which can be caused by a loss of the chunk in the channel K or a loss of the acknowledgement in the channel L). As pointed out in [MV93], if we allow acknowledgements to come after T1 times out, the protocol may exhibit wrong behaviours leading to the "silent" loss of data chunks. For example, suppose that a chunk c_1 sent by S is correctly received by R and acknowledged, but T1 times out before the acknowledgement arrives. The chunk c_1 is then resent by S and it is again correctly received and acknowledged by R, which interprets it as a repetition. Now, the first acknowledgement arrives, S associates it to the second transmission of c_1 and sends the following chunk c_2. Suppose now that c_2 is lost in the channel K and the second acknowledgement arrives normally (i.e., before T1 times out); S interprets it as a correct transmission of the chunk c_2 and sends a new chunk. To solve this problem, we adopt the approach followed by Groote and van de Pol [GvdP93], and we model this causality constraint by allowing the channels K and L to synchronize with the timer T1 via an auxiliary gate LOST. Once started, T1 is enabled to time out only after a signal is sent by one of the channels at the LOST gate, meaning the loss of the data chunk or of the corresponding acknowledgement. This approach is also used by Helmink and al. in their BRP modeling [HSV94].

Another synchronization link, which will be explained later, is added between the sender and the T2 timer via the SYNC gate, in order to properly restart the sender and the receiver when the protocol gives up the transmission of a packet.

These auxiliary synchronization links LOST and SYNC allow a correct modeling of the BRP protocol assuming that the values are properly set for the two timers. Under these conditions, we know from [HSV94] that the BRP protocol can be modeled as a time-independent system. If these conditions do not hold, then the behaviour of the BRP protocol becomes time-dependent and one should use a timed formalism to model and analyze the BRP in presence of explicit time delays.

The architecture of the BRP protocol is described in LOTOS by the behaviour expression below. The entities on Figure 1 are modeled by LOTOS processes which execute concurrently. We are interested only in the INPUT and OUTPUT actions of the protocol and, therefore, we hide all the other gates using the "**hide**" operator. We use the parallel asynchronous operator "|||" to compose the processes which do not synchronize directly (the sender and the receiver, the two communication channels, and the two timers) and the "|[...]|" operator to connect the synchronizing groups of processes via the appropriate gates. The sender and the receiver synchronize with the two communication channels on the SEND_K, REC_K, SEND_L, and REC_L gates. The two timers synchronize with the sender, the receiver and the channels on the T1, T2, LOST, and SYNC gates.

```
hide T1, T2, SEND_K, REC_L, REC_K, SEND_L, LOST, SYNC in
   (
      (
         (
         S [INPUT, SEND_K, REC_L, T1, SYNC] (false)
         |||
         R [OUTPUT, REC_K, SEND_L, T2]
         )
      |[SEND_K, REC_K, SEND_L, REC_L]|
         (
         K [SEND_K, REC_K, LOST]
         |||
         L [SEND_L, REC_L, LOST]
         )
      )
   |[T1, LOST, T2, SYNC]|
      (
         T1 [T1, LOST]
         |||
         T2 [T2, SYNC]
      )
   )
```

In the next sections, we detail each of these six processes.

4.2.2 The sender

The sender is described by the process S below. It reads a data packet from the sender client at the INPUT gate and, if the packet is not empty, calls the S_1 process, which handles the transmission of the packet. The ALT parameter represents the current value of the so-called *alternating bit* used by the protocol to detect duplication of the data chunks.

```
process S [INPUT, SEND_K, REC_L, T1, SYNC] (ALT:Bool) : noexit :=
   INPUT ?P:Packet;
      (
      [len (P) == 0] -> (* empty packets rejected *)
         S [INPUT, SEND_K, REC_L, T1, SYNC] (ALT)
      []
      [len (P) > 0] ->
         S_1 [INPUT, SEND_K, REC_L, T1, SYNC] (ALT, P, len (P), 0)
      )
endproc
```

Process S_1 scans the data packet P and sends the data chunks, one by one, on the channel K via the SEND_K gate. Each chunk is accompanied by three bits (modeled here as boolean values): the first two bits indicate whether the chunk is the first and/or the last of the packet, and the third bit is the alternating bit. The value parameter L stands for the initial length of the packet. RN denotes the current number of retransmission attempts. The timer T1 is started before sending each chunk. Two situations are possible.

In the first case, an acknowledgement arrives on the REC_L gate before T1 has expired. The sender resets the timer T1 and continues to transmit the packet. This is modeled by a recursive call to process S_1 with the packet P truncated to its remaining part tail (P), the retransmission counter reset[10] to 0, and the alternating bit switched. If the chunk was the last one of the packet, a confirmation I_OK is issued to the sending client and the sender returns to its initial state by a call to process S.

In the second case, timer T1 times out, meaning a transmission failure on one of the channels. To avoid wrong behaviours of the protocol (such as those mentioned in Section 4.2.1), we must ensure that an acknowledgement cannot arrive at the sender after the timer T1 has expired. Practically, this can be achieved by setting the timeout value for T1 to the round-trip transmission time t (i.e., the time required for the transmission of a chunk and of its acknowledgement). If the RN counter is inferior to the maximum number max of retransmission attempts, the current chunk is retransmitted by a call to process S_1 with the retransmission counter incremented. If the max limit is reached, the sender aborts the transmission of the current packet by sending an I_NOK or I_DK confirmation to the sending client; before accepting another packet, the sender must ensure that the receiver has

[10]We follow here again the Helmink and al. description. In the Groote and van de Pol's one, the retransmission counter is not reset after the successful transmission of a chunk.

also detected this situation in order to properly restart the whole protocol. Practically, this can be guaranteed by setting the timeout value for the T2 timer to a value greater than $\texttt{max} * t$, where t is the round-trip transmission time. Since real-time aspects cannot be directly modeled in LOTOS, we use instead an auxiliary synchronization on the SYNC gate to ensure that timer T2 does not time out until the sender has given up the transmission of the packet. Thus, the sender issues a "sender ready" signal (modeled as a value S_READY of an enumerated type) on the SYNC gate, enabling the T2 timer to time out, waits for a "receiver ready" (R_READY) reply on the same gate, and then returns to its initial state by calling the S process.

Notice that in all the calls to process S, the ALT bit is switched and thus the alternating bit scheme is continued for the subsequent packets.

```
process S_1 [INPUT, SEND_K, REC_L, T1, SYNC]
            (ALT:Bool, P:Packet, L, RN:Nat) : noexit :=
   T1 !START;
      SEND_K !(len(P) == L) !(len(P) == 1) !ALT !head(P);
      (
      REC_L;
         T1 !RESET;
            (
            [len (P) == 1] ->
               INPUT !I_OK;
                  S [INPUT, SEND_K, REC_L, T1, SYNC] (not (ALT))
            []
            [len (P) > 1] ->
               S_1 [INPUT, SEND_K, REC_L, T1, SYNC] (not (ALT), tail (P), L, 0)
            )
      []
      T1 !TIMEOUT;
         (
         [RN < max] ->
            S_1 [INPUT, SEND_K, REC_L, T1, SYNC] (ALT, P, L, RN + 1)
         []
         [RN == max] ->
            INPUT !conf (P);
               SYNC !S_READY;
                  SYNC !R_READY;
                     S [INPUT, SEND_K, REC_L, T1, SYNC] (not (ALT))
         )
      )
endproc
```

The timer associated to the sender is modeled by the process T1 below. It is started by a START signal on the T1 gate. Then, if it receives a RESET signal at the same gate, it returns to its initial state (using a recursive call of process T1); if it receives a loss indication from one of the channels at the LOST gate, it issues a TIMEOUT signal at the T1 gate and returns to its initial state.

```
process T1 [T1, LOST] : noexit :=
   T1 !START;
      (
      T1 !RESET;
```

```
            T1 [T1, LOST]
         []
         LOST; (* loss indication *)
            T1 !TIMEOUT;
               T1 [T1, LOST]
         )
   endproc
```

4.2.3 The communication channels

The unreliable communication channel K is modeled as a lossy one-slot buffer that repeatedly reads a data chunk (accompanied by the three control bits) at the SEND_K gate, eventually transmits it at the REC_K gate and returns to its initial state. After reading a data chunk, the channel K may lose it (i.e., do not transmit it at the REC_K gate) and indicate this loss by sending a signal at the LOST gate. The choice between transmitting or losing a chunk is preceded by silent actions "i" to ensure that the environment cannot influence the decision.

```
process K [SEND_K, REC_K, LOST] : noexit :=
   SEND_K ?FST, LST, ALT:Bool ?D:Data;
      (
      i;
         REC_K !FST !LST !ALT !D; (* correct transmission *)
            K [SEND_K, REC_K, LOST]
      []
      i;
         LOST;                    (* loss indication *)
            K [SEND_K, REC_K, LOST]
      )
endproc
```

The channel L exhibits a similar behaviour, except that it receives acknowledgements at the SEND_L gate (instead of SEND_K) and transmits them at the REC_L gate (instead of REC_K).

```
process L [SEND_L, REC_L, LOST] : noexit :=
   SEND_L;
      (
      i;
         REC_L;                   (* correct transmission *)
            L [SEND_L, REC_L, LOST]
      []
      i;
         LOST;                    (* loss indication *)
            L [SEND_L, REC_L, LOST]
      )
endproc
```

4.2.4 The receiver

The receiver is modeled by the process R below. It waits for an incoming data chunk (the first one of a new packet) at the REC_K gate, delivers it to the receiving client at the OUTPUT gate (together with an I_FST or I_OK indication), sends an acknowledgement to the channel L and calls the process R_1, which handles the reception of the current packet. The timer T2 is started each time an acknowledgement is sent at the SEND_L gate.

```
process R [OUTPUT, REC_K, SEND_L, T2] : noexit :=
   REC_K ?FST, LST, ALT:Bool ?D:Data [FST];
      OUTPUT !D !ind (FST, LST);
         T2 !START;
            SEND_L;
               R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, not (ALT))
endproc
```

The END parameter of process R_1 indicates whether the last chunk delivered was the last one of the current packet or not. The ALT parameter is the alternating bit expected by the receiver for the next data chunk.

After sending an acknowledgement, several situations can occur.

In the first case, a new data chunk (i.e., a chunk with an alternating bit equal to ALT) is received at the REC_K gate before T2 has expired. Then, timer T2 is reset, the chunk is delivered to the receiving client with the appropriate indication (notice that it may be the first chunk of a new packet), an acknowledgement is sent to the channel L, timer T2 is started again and the reception is continued by a recursive call of the process R_1.

In the second case, a duplicated chunk is received at the REC_K gate before T2 has timed out. Then, the chunk is simply discarded and acknowledged (without resetting the timer T2), and the reception is continued.

In the third case, a timeout signal is received from the T2 timer, meaning the loss of contact with the sender. If the current packet has not been completely delivered, an I_NOK indication is issued to the receiving client. An R_READY signal is sent to T2 in response to the timeout, and the receiver returns to its initial state by calling the process R.

```
process R_1 [OUTPUT, REC_K, SEND_L, T2] (END, ALT:Bool) : noexit :=
   REC_K ?FST, LST:Bool !ALT ?D:Data;       (* new chunk *)
      T2 !RESET;
         OUTPUT !D !ind (FST, LST);
            T2 !START;
               SEND_L;
                  R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, not (ALT))
   []
   REC_K ?FST, LST:Bool !not (ALT) ?D:Data; (* duplicated chunk *)
      SEND_L;
         R_1 [OUTPUT, REC_K, SEND_L, T2] (LST, ALT)
   []
   T2 !TIMEOUT;
      (
      [not (END)] ->                         (* more chunks to follow *)
         OUTPUT !I_NOK;
```

```
                T2 !R_READY;
                   R [OUTPUT, REC_K, SEND_L, T2]
          []
          [END] ->                            (* last chunk already delivered *)
             T2 !R_READY;
                R [OUTPUT, REC_K, SEND_L, T2]
          )
    endproc
```

The timer T2 has a more complicated behaviour than T1. If it is started, it can be reset (normal functioning of the protocol) or it can receive an S_READY signal on the SYNC gate, meaning that the sender has given up the transmission and "enables" T2 to time out. After a TIMEOUT signal, in order to ensure synchronization of the sender and the receiver for a new transmission, T2 waits for an R_READY signal on the T2 gate and retransmits it to the sender via the SYNC gate. If we do not model this synchronization, the protocol may exhibit wrong behaviours leading again to the silent loss of data chunks. For example, suppose that a chunk c_0 with an alternating bit 0 is successfully received and acknowledged by R (which also starts T2), but the next chunk c_1 (with an alternating bit 1) is systematically lost by the channel K such that S aborts the transmission of the packet. The sender can then read another data packet and transmit its first chunk, say c_0', with an alternating bit 0 (since the alternating bit scheme is continued between subsequent data packets). Now, if the timer T2 does not time out before c_0' arrives at R, the receiver will interpret the chunk as a repetition (since the last chunk received was c_0), will reject it and send an acknowledgement to S, which will take it as a correct transmission of c_0'. Then, the protocol can continue its normal behaviour without detecting the loss of the chunks c_1 and c_0'.

The modeling of the causal constraints using synchronizations on the SYNC gate makes possible that an S_READY signal arrives at the SYNC gate even if T2 is not started (for example, this happens when the first chunk of a packet is systematically lost by the channel K and the sender gives up the transmission). To react properly to such event, timer T2 must respond immediately with an R_READY signal at the SYNC gate, informing the sender that it can safely begin the transmission of a new data packet.

```
    process T2 [T2, SYNC] : noexit :=
       T2 !START;
          (
          T2 !RESET;
             T2 [T2, SYNC]
          []
          SYNC !S_READY;
             T2 !TIMEOUT;
                T2 !R_READY;
                   SYNC !R_READY;
                      T2 [T2, SYNC]
          )
       []
       SYNC !S_READY;
          SYNC !R_READY;
             T2 [T2, SYNC]
    endproc
```

5 Model generation

In order to perform verifications by model-checking, we first generated the LTSs corresponding to the LOTOS descriptions of the BRP protocol and service using the CÆSAR and CÆSAR.ADT compilers.

Before presenting the experimental results concerning the generation of the two models, we give the formal definition of an LTS corresponding to a LOTOS program.

5.1 The LTS of a LOTOS program

According to the operational semantics of LOTOS [ISO88b], each LOTOS description can be translated into a (possibly infinite) LTS, which encodes all its possible execution sequences. An LTS is formally defined as a 4-uple $M = \langle Q, A, T, q_{init} \rangle$ where:

- Q is the set of *states* of the program;
- A is a set of *actions* performed by the program. An action $a \in A$ is a tuple $G\ V_1, ..., V_n$ where G is a *gate* and V_1, ..., V_n $(n \geq 0)$ are the values exchanged (i.e., sent or received) during the rendez-vous at G. For the *silent* action τ, the value list must be empty $(n = 0)$;
- $T \subseteq Q \times A \times Q$ is the *transition relation.* A transition $\langle q_1, a, q_2 \rangle \in T$ (written also "$q_1 \xrightarrow{a} q_2$") means that the program can move from state q_1 to state q_2 by performing action a;
- $q_{init} \in Q$ is the *initial state* of the program.

For each state $q \in Q$, we note $Path(q)$ the set of all paths $q(= q_0) \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 ...$ issued from q.

5.2 Generation of the BRP protocol and service LTSs

The LOTOS descriptions of the BRP protocol and service contain three parameters with infinite data domains: the length of the data packets (type `Packet`), the nature of the data chunks (type `Data`) and the maximum number `max` of retransmission attempts (type `Nat`). To generate finite LTSs in order to perform verifications by model-checking, we must restrict these domains to finite sets.

We achieve this in the following way. First, to obtain data packets of finite length, we connect the protocol and service descriptions to a particular environment (modeled as a particular sending client), which produces data packets having a maximum fixed length. Second, since communication channels are modeled as one-place buffers and preserve message ordering, we may identify a data chunk by its position in the packet and do not care about its contents[11]. Third, for each experiment, we fix a value for the maximum number of retransmissions.

We made two series of experiments, all of them performed on a Sparc 10 machine with 64 Mbytes of memory.

In the first series of experiments, we added to both LOTOS descriptions an auxiliary process modeling a sending client, which repeatedly sends a data packet of length 20 at the `INPUT` gate, waits for a confirmation from the sender entity of the protocol, and returns to its initial state:

```
process SENDING_CLIENT [INPUT] : noexit :=
   INPUT !cons_packet (20); (* send the packet *)
```

[11] In Groote and van de Pol's μCRL description, the data chunks have a constant value d_0.

```
         INPUT ?IO:Ind;        (* receive the acknowledgement *)
            SENDING_CLIENT [INPUT]
   endproc
```

The auxiliary function `cons_packet (N)` (see Annex A) constructs a data packet of `N` chunks respectively numbered from 1 to `N`: `data (1)`, `data (2)`, ..., `data (N)`.

Then, using the CÆSAR and CÆSAR.ADT compilers, we generated the LTSs corresponding to the two LOTOS descriptions for a maximum number of retransmissions ranging between 0 and 10.

The LTS of the BRP service has 180 states and 240 transitions and was generated in about 5 seconds. Its size is independent from the number of retransmissions, since this "implementation detail" is not relevant to the definition of the service.

The results concerning the LTS of the BRP protocol are shown in Table 1. For each experiment, the table gives the size (in number of states and transitions) of the LTS and the time (in minutes) required for its generation. We remark a linear variation of the number of states and transitions, and a quasi-linear variation of the generation time in function of the maximum number of retransmissions.

retrans. number	*protocol*		*time*
	states	*trans.*	
0	3,927	4,309	0′27″
1	13,765	15,757	1′37″
2	19,329	22,563	3′20″
3	24,893	29,369	5′49″
4	30,457	36,175	8′49″
5	36,021	42,981	12′38″
6	41,585	49,787	16′58″
7	47,149	56,593	21′18″
8	52,713	63,399	26′20″
9	58,277	70,205	31′19″
10	63,841	77,011	36′14″

Table 1: Generation of the protocol LTS for data packets of length 20

In the second series of experiments, we considered a more general sending client, which repeatedly produces data packets of random length between 1 and 10:

```
   process SENDING_CLIENT [INPUT] : noexit :=
      choice N:Nat []
         [(1 <= N) and (N <= 10)] ->
            INPUT !cons_packet (N); (* send the packet *)
               INPUT ?IO:Ind;       (* receive the acknowledgement *)
                  SENDING_CLIENT [INPUT]
   endproc
```

The "**choice**" operator followed by the boolean guard arbitrarily chooses the value `N` between 1 and 10.

retrans.	*protocol*		*time*
number	*states*	*trans.*	
0	140,511	197,478	1 : 32′48″
1	417,379	565,050	4 : 23′52″
2	543,725	724,460	5 : 38′06″
3	670,071	883,870	6 : 52′20″

Table 2: Generation of the protocol Lts for data packets with random lengths between 1 and 10

The Lts of the Brp service generated in this case has 981 states and 2145 transitions, and was generated in about 10 seconds.

Table 2 shows the results concerning the generation of the protocol Lts (times being measured in hours) for a maximum number of retransmissions ranging from 0 to 3.

6 Verification using bisimulations

After generating the Ltss corresponding to the Brp protocol and service, we performed verification by means of bisimulations using the Aldébaran tool.

For each experiment presented in Section 5.2, we checked that the Lts of the protocol is equivalent to the Lts of the service modulo branching bisimulation [vGW89].

In order to reduce the verification time, we first minimized both Ltss modulo strong equivalence [Par81] before comparing them modulo branching equivalence. All verifications have been performed on a Sparc 10 machine with 64 Mbytes of memory.

Table 3 presents the verification results for the first series of experiments described in Section 5.2. For each experiment, the table gives the reduction and the comparison times (in minutes) obtained using Aldébaran. The protocol is branching equivalent to its service even if we do not allow any

retrans.	*bisimulation times*		
number	*reduction*	*comparison*	*total*
0	0′02″	0′01″	0′03″
1	0′04″	0′03″	0′07″
2	0′08″	0′06″	0′14″
3	0′10″	0′09″	0′19″
4	0′14″	0′11″	0′25″
5	0′17″	0′15″	0′32″
6	0′20″	0′17″	0′37″
7	0′24″	0′19″	0′43″
8	0′26″	0′22″	0′48″
9	0′29″	0′24″	0′53″
10	0′32″	0′30″	1′02″

Table 3: Verification using bisimulations for data packets of length 20

retransmission attempt. We remark that the reduction time is slightly superior to the comparison time, and both have a quasi-linear variation with the number of retransmissions.

Table 4 shows the same verification results for the second series of experiments described in Section 5.2. We remark again a quasi-linear variation of the reduction and comparison times with the number of retransmissions.

retrans.	*bisimulation times*		
number	*reduction*	*comparison*	*total*
0	1′19″	0′08″	1′27″
1	3′45″	0′53″	4′38″
2	4′09″	1′33″	5′42″
3	5′44″	2′28″	8′12″

Table 4: Verification using bisimulations for data packets with random lengths between 1 and 10

For both series of experiments, the total time required for verification by means of bisimulations is significantly smaller than the time required for LTS generation.

7 Verification using temporal logics

As an alternative to verification using bisimulations, we also performed verification using temporal logics.

As LOTOS dynamic semantics is action-based, it is natural to choose a temporal logic interpreted over the actions of the LTS model. In this case-study we used a simplified fragment of the ACTL (Action CTL) temporal logic defined in [NV90], which is sufficiently powerful to express safety and liveness properties.

The following sections present a short description of the syntax and semantics of the ACTL fragment we used, and the verification results of the BRP safety and liveness properties expressed in ACTL.

7.1 The ACTL temporal logic

In order to express predicates over the program actions (the so-called *basic predicates*), a small auxiliary logic of actions is needed. The action formulas α of this logic have the following syntax:

$$\begin{array}{rcl} \alpha & ::= & \mathbf{true} \\ & | & \{G\ V_1, ..., V_n\} \\ & | & \neg\alpha \\ & | & \alpha\ \wedge\ \alpha' \end{array}$$

The construction $\{G\ V_1, ..., V_n\}$ denotes an *action pattern*, where G is a gate name and the values V_i $(1 \leq i \leq n,\ n \geq 0)$ match the corresponding values exchanged (i.e., sent or received) when the action is performed. For simplicity purposes, and unlike the original ACTL logic, we also allow action patterns (of the form $\{\tau\}$) matching τ-actions.

Of course, the usual derived boolean operators are also allowed: we write **false** for $\neg$ **true**, $\alpha\ \vee\ \alpha'$ for $\neg(\neg\alpha\ \wedge\ \neg\alpha')$, and $\alpha \Longrightarrow \alpha'$ for $\neg\alpha\ \vee\ \alpha'$.

The action formulas α are interpreted over the actions $a \in A$ of the model $M = \langle Q, A, T, q_{init}\rangle$ corresponding to a LOTOS program. The satisfaction of an action formula α by an action $a \in A$, written $a \models_M \alpha$ (or simply $a \models \alpha$ if the model M is understood), is defined inductively by:

$$\begin{array}{ll} a \models \mathbf{true} & \text{always;} \\ a \models \{G\ V_1, ..., V_n\} & \text{iff } a = G\ V_1, ..., V_n; \\ a \models \neg\alpha & \text{iff } a \not\models \alpha; \\ a \models \alpha\ \wedge\ \alpha' & \text{iff } a \models \alpha \text{ and } a \models \alpha'. \end{array}$$

The formulas φ of the ACTL fragment we used are defined by the following syntax:

$$\begin{array}{lcl} \varphi & ::= & \mathbf{true} \\ & | & \neg\varphi \\ & | & \varphi\ \wedge\ \varphi' \\ & | & \mathbf{EX}_\alpha\varphi \\ & | & \mathbf{AX}_\alpha\varphi \\ & | & \mathbf{E}\left[\varphi_\alpha\mathbf{U}\varphi'\right] \\ & | & \mathbf{A}\left[\varphi_\alpha\mathbf{U}\varphi'\right] \end{array}$$

The satisfaction of an ACTL formula φ by a state $q \in Q$ of an LTS $M = \langle Q, A, T, q_{init}\rangle$, written $q \models_M \varphi$ (or simply $q \models \varphi$ if the model M is understood), is defined inductively by:

$$\begin{array}{lll} q \models \mathbf{true} & & \text{always;} \\ q \models \neg\varphi & \text{iff} & q \not\models \varphi; \\ q \models \varphi\ \wedge\ \varphi' & \text{iff} & q \models \varphi \text{ and } q \models \varphi'; \\ q \models \mathbf{EX}_\alpha\varphi & \text{iff} & \exists q \xrightarrow{a} q' \in T \text{ such that } a \models \alpha \text{ and } q' \models \varphi; \\ q \models \mathbf{AX}_\alpha\varphi & \text{iff} & \forall q \xrightarrow{a} q' \in T,\ a \models \alpha \text{ and } q' \models \varphi; \\ q \models \mathbf{E}\left[\varphi_\alpha\mathbf{U}\varphi'\right] & \text{iff} & \exists q (= q_0) \xrightarrow{a_0} q_1 \xrightarrow{a_1} ... \in \mathit{Path}(q), \\ & & \exists k > 0 \text{ such that } q_k \models \varphi' \text{ and } \forall i \in [0; k-1], q_i \models \varphi \text{ and } a_i \models \alpha; \\ q \models \mathbf{A}\left[\varphi_\alpha\mathbf{U}\varphi'\right] & \text{iff} & \forall q (= q_0) \xrightarrow{a_0} q_1 \xrightarrow{a_1} ... \in \mathit{Path}(q), \\ & & \exists k > 0 \text{ such that } q_k \models \varphi' \text{ and } \forall i \in [0; k-1], q_i \models \varphi \text{ and } a_i \models \alpha. \end{array}$$

A model $M = \langle Q, A, T, q_{init}\rangle$ satisfies a formula φ, noted $M \models \varphi$ (or simply φ if the model M is understood), if and only if $q \models \varphi$ for all $q \in Q$.

The usual derived modalities are defined as follows:

$$\begin{array}{lcl} \langle\alpha\rangle\,\varphi & = & \mathbf{EX}_\alpha\varphi \\ \left[\alpha\right]\varphi & = & \neg\,\langle\alpha\rangle\,\neg\varphi \\ \mathbf{EF}_\alpha\varphi & = & \mathbf{E}\left[\mathbf{true}_\alpha\mathbf{U}\varphi\right] \\ \mathbf{AF}_\alpha\varphi & = & \mathbf{A}\left[\mathbf{true}_\alpha\mathbf{U}\varphi\right] \\ \mathbf{EG}_\alpha\varphi & = & \neg\mathbf{AF}_\alpha\neg\varphi \\ \mathbf{AG}_\alpha\varphi & = & \neg\mathbf{EF}_\alpha\neg\varphi \end{array}$$

The $\langle\alpha\rangle\,\varphi$ and $[\alpha]\,\varphi$ operators are the well-known Hennessy-Milner modalities [HM85]. A state q satisfies $\langle\alpha\rangle\,\varphi$ (*resp.* $[\alpha]\,\varphi$) if some (*resp.* all) of its direct successors reached after an action satisfying α satisfies (*resp.* satisfy) φ. A state q satisfies $\mathbf{EF}_\alpha\varphi$ (*resp.* $\mathbf{AF}_\alpha\varphi$) iff some path (*resp.* all paths) issued from q leads (*resp.* lead) via actions satisfying α to a state satisfying φ. A state q satisfies $\mathbf{EG}_\alpha\varphi$ (*resp.* $\mathbf{AG}_\alpha\varphi$) iff for some path (*resp.* all paths) issued from q, every prefix consisting of actions that satisfy α leads to a state satisfying φ.

To verify the BRP protocol, we defined in ACTL a set of 21 safety and liveness properties, which, we expect, characterize the essential functioning properties of the protocol (although it is notoriously difficult to find a minimal set of temporal logic formulas that completely characterize the behaviour of a parallel program).

We decided to limit the size of the data packets to 3 chunks, because this allows to take in account all the interesting cases of the BRP protocol behaviour: all the types of indications and confirmations (`I_FST`, `I_INC`, `I_OK`, `I_NOK`, `I_DK`) issued by the protocol to the sending and receiving clients are present.

To express formulas about data packets of arbitrary lengths, one should use a more powerful formalism than the ACTL temporal logic, for example the modal μ-calculus [Koz83].

7.2 Safety properties

Informally, a safety property of a concurrent program specifies that "something bad never happens."

All the safety properties we exhibited for the BRP protocol have a common form. Therefore, we define a "**not** α **to** $\alpha_1, \alpha_2, ..., \alpha_n$ **unless** α'" shorthand notation, meaning that after an action satisfying α it is not possible to reach and execute a sequence of actions satisfying $\alpha_1, \alpha_2, ..., \alpha_n$ without performing an action which satisfies α'. This can be expressed in ACTL as follows:

$$\begin{array}{l} \textbf{not } \alpha \textbf{ to } \alpha_1, \alpha_2, ..., \alpha_n \textbf{ unless } \alpha' = \\ [\alpha]\, \neg \mathbf{EF}_{\neg\alpha'}\, \langle\alpha_1\rangle\, \mathbf{EF}_{\mathbf{true}}\, \langle\alpha_2\rangle \,...\mathbf{EF}_{\mathbf{true}}\, \langle\alpha_n\rangle\, \mathbf{true} \end{array}$$

For conciseness, we will use the "**not** ... **to** ... **unless**" notation throughout this Section.

Also, for clarity, we will use suggestive names for the action patterns in the α basic predicates instead of the precise syntax defined in Section 7.1. For example, instead of the action pattern "`{ INPUT cons (data (1), nil) }`", which characterizes the action of reading a one-chunk packet at the `INPUT` gate, we will simply write "`IN_d1`."

First, we examine the behaviour from the sending client point of view. A sending client has the following cyclical behaviour: it sends a data packet to the sender entity of the protocol (action `IN_d1_dn`), waits for an `I_OK`, `I_NOK` or `I_DK` confirmation (action `IN_CONF`), and returns to its initial state. This alternation between the `IN_d1_dn` and `IN_CONF` actions can be expressed by the two ACTL formulas below:

not `IN_d1_dn` **to** `IN_d1_dn` **unless** `IN_CONF`

not `IN_CONF` **to** `IN_CONF` **unless** `IN_d1_dn`

In a similar way, the receiving client must observe an alternation between the chunks marking the packet beginnings (actions `OUT_d1_FST`) and the chunks (actions `OUT_dn_OK`) or indications (actions `OUT_NOK`) delimiting packet endings. The data packets of length one must be treated as a particular case, since the reception of the single chunk (action `OUT_d1_OK`) marks both the beginning and the ending of the packet. The proper orderings of actions at the receiving client are specified by the following ACTL formulas:

not `OUT_d1_FST` **to** `OUT_d1_FST` **unless** `OUT_dn_OK` $\vee$ `OUT_NOK`

not `OUT_dn_OK` $\vee$ `OUT_NOK` **to** `OUT_dn_OK` $\vee$ `OUT_NOK` **unless** `OUT_d1_FST`

not `OUT_d1_FST` **to** `OUT_d1_OK` **unless** `OUT_dn_OK` $\vee$ `OUT_NOK`

We examine now the properties concerning the sequencing of the actions at both sending client and receiving client sides.

After accepting a data packet, the protocol cannot issue an `I_OK` confirmation (action `IN_OK`) to the sending client unless the same indication (action `OUT_dn_OK`) has been delivered to the receiving client:

not `IN_d1_dn` **to** `IN_OK` **unless** `IN_d1_dn` $\vee$ `OUT_dn_OK`

Here (and also for the subsequent formulas in this section) we added the `IN_d1_dn` action predicate to the "**unless**" clause to ensure that the property deals with the same data packet (no other packet has been accepted meanwhile from the sending client).

After accepting a data packet, the protocol cannot issue an `I_NOK` indication to the receiving client (action `OUT_NOK`) before issuing an `I_NOK` or `I_DK` confirmation (action `IN_NOK`, *resp.* `IN_DK`) to the sending client:

$$\textbf{not}\ \texttt{IN\_d1\_dn}\ \textbf{to}\ \texttt{OUT\_NOK}\ \textbf{unless}\ \texttt{IN\_d1\_dn} \lor \texttt{IN\_NOK} \lor \texttt{IN\_DK}$$

After a one-chunk data packet has been accepted from the sending client (action `IN_d1`), the receiving client cannot get an `I_NOK` indication:

$$\textbf{not}\ \texttt{IN\_d1}\ \textbf{to}\ \texttt{OUT\_NOK}\ \textbf{unless}\ \texttt{IN\_d1\_dn}$$

After accepting a data packet of length 2 (action `IN_d1_d2`), the receiving client cannot get the second chunk successfully (action `OUT_d2_OK`) before receiving the first chunk (action `OUT_d1_FST`):

$$\textbf{not}\ \texttt{IN\_d1\_d2}\ \textbf{to}\ \texttt{OUT\_d2\_OK}\ \textbf{unless}\ \texttt{IN\_d1\_dn} \lor \texttt{OUT\_d1\_FST}$$

After accepting a data packet of length 3 (action `IN_d1_d2_d3`), the receiving client cannot successfully get the third chunk (action `OUT_d3_OK`) before receiving the second chunk (action `OUT_d2_INC`):

$$\textbf{not}\ \texttt{IN\_d1\_d2\_d3}\ \textbf{to}\ \texttt{OUT\_d3\_OK}\ \textbf{unless}\ \texttt{IN\_d1\_dn} \lor \texttt{OUT\_d2\_INC}$$

Finally, after accepting a data packet of length 3, the protocol cannot deliver the second and third chunk to the receiving client before delivering the first one:

$$\textbf{not}\ \texttt{IN\_d1\_d2\_d3}\ \textbf{to}\ \texttt{OUT\_d2\_INC},\ \texttt{OUT\_d3\_OK}\ \textbf{unless}\ \texttt{IN\_d1\_dn} \lor \texttt{OUT\_d1\_FST}$$

7.3 Liveness properties

Informally, a liveness property of a concurrent program specifies that "something good eventually happens."

The liveness properties that we exhibited for the Brp protocol deal, on one hand, with the reachability of certain actions, and on the other hand, with the system responses to certain actions.

An important reachability property is that from every state of the system, it is inevitable to reach and execute the action of reading a data packet from the sending client (action `IN_d1_dn`):

$$\mathbf{AG}_{\mathbf{true}}\ \mathbf{AF}_{\mathbf{true}}\ \mathbf{AX}_{\texttt{IN\_d1\_dn}}\ \mathbf{true}$$

This is a powerful liveness property, which implies both deadlock freedom and reachability of `IN_d1_dn` actions independently from the (fair or unfair) scheduling of actions.

Besides this property, we can examine the responses of the system after reading a data packet.

We start by looking at the sender side. The sending client always gets an `I_OK`, `I_NOK` or `I_DK` confirmation (action `IN_CONF`) after sending a data packet:

$$[\texttt{IN\_d1\_dn}]\ \mathbf{AF}_{\neg\texttt{IN\_d1\_dn}}\ \mathbf{AX}_{\texttt{IN\_CONF}}\ \mathbf{true}$$

Here, we added the $\neg$`IN_d1_dn` clause to the **AF** operator to ensure that the formula deals with the same data packet (no other packet has been read until the `IN_CONF` action).

We consider now the messages issued at both the sender and receiver sides in response to the actions performed by the sending and receiving clients. Each property given below has two parts.

The first part states the fact that, after an action sequence has occurred, some response is eventually reached and executed. We can express this using an "**after** $\alpha_1, \alpha_2, ..., \alpha_n$ **inev** α" shorthand notation, meaning that after a sequence of actions satisfying $\alpha_1, \alpha_2, ..., \alpha_n$ (possibly separated by τ-actions), it is inevitable to execute (maybe via some τ-actions) an action satisfying α. This can be translated in ACTL as follows:

$$\begin{array}{l}\textbf{after } \alpha_1, \alpha_2, ..., \alpha_n \textbf{ inev } \alpha = \\ [\alpha_1]\,\mathbf{AG}_\tau\,[\alpha_2]\,...\mathbf{AG}_\tau\,[\alpha_n]\,\mathbf{AF}_\tau\ \mathbf{AX}_\alpha\ \mathbf{true}\end{array}$$

The second part states that the action sequence causing the desired response exist in the model. This can be expressed using an "**after** α **pot** $\alpha_1, \alpha_2, ..., \alpha_n$" shorthand notation, meaning that after an action satisfying α, it is possible to execute (maybe via some τ-actions) a sequence of actions (possibly separated by τ-actions) that satisfy $\alpha_1, \alpha_2, ..., \alpha_n$. The ACTL translation is the following:

$$\begin{array}{l}\textbf{after } \alpha \textbf{ pot } \alpha_1, \alpha_2, ..., \alpha_n = \\ [\alpha]\,\mathbf{EF}_\tau\,\langle\alpha_1\rangle\,\mathbf{EF}_\tau\,\langle\alpha_2\rangle\,...\mathbf{EF}_\tau\,\langle\alpha_n\rangle\,\mathbf{true}\end{array}$$

For conciseness, we will use these two notations in the sequel.

After a data packet of length 3 has been accepted (action `IN_d1_d2_d3`) and an `I_NOK` confirmation (action `IN_NOK`) has been issued to the sending client (meaning a loss of the first chunk), the protocol eventually reads a new packet without issuing any indication to the receiving client:

after `IN_d1_d2_d3`, `IN_NOK` **inev** `IN_d1_dn`

after `IN_d1_d2_d3` **pot** `IN_NOK`

After a data packet of length 3 has been read, the first chunk has been successfully delivered to the receiving client (action `OUT_d1_FST`), and an `I_NOK` confirmation has been received at the sending client (meaning the loss of the second chunk), the protocol eventually issues an `I_NOK` indication to the receiving client:

after `IN_d1_d2_d3`, `OUT_d1_FST`, `IN_NOK` **inev** `OUT_NOK`

after `IN_d1_d2_d3` **pot** `OUT_d1_FST`, `IN_NOK`

After a data packet of length 3 has been read, the first two chunks have been delivered to the receiving client (actions `OUT_d1_FST` and `OUT_d2_INC`), and an `I_DK` confirmation has been received at the sending client (meaning the loss of the third chunk), the protocol eventually informs the receiving client via an `I_NOK` indication:

after `IN_d1_d2_d3`, `OUT_d1_FST`, `OUT_d2_INC`, `IN_DK` **inev** `OUT_NOK`

after `IN_d1_d2_d3` **pot** `OUT_d1_FST`, `OUT_d2_INC`, `IN_DK`

Finally, after a data packet of length 3 has been accepted and all the chunks have been successfully delivered to the receiving client, the sending client eventually gets either an `I_DK` confirmation (meaning the loss of the last acknowledgement) or an `I_OK` confirmation (meaning a successful transmission of the packet):

after `IN_d1_d2_d3`, `OUT_d1_FST`, `OUT_d2_INC`, `OUT_d3_OK` **inev** `IN_DK` $\vee$ `IN_OK`

after `IN_d1_d2_d3` **pot** `OUT_d1_FST`, `OUT_d2_INC`, `OUT_d3_OK`

7.4 Verification

The 21 safety and liveness properties given in Sections 7.2 and 7.3 have been verified on the LTS corresponding to the BRP protocol using the XTL [Mat94] prototype model-checker.

It is worth noticing that, since the XTL language allows the definition of macro-notations (for both basic predicates and temporal operators), the ACTL formulas given in Sections 7.2 and 7.3 are almost identical to those written in the XTL source code.

The LTS of the BRP protocol was generated using CÆSAR (for data packets of random lengths between 1 and 3 and a maximum number of 5 retransmissions) and then minimized with ALDÉBARAN modulo strong equivalence. We checked the properties on the reduced LTS, which had 457 states and 559 transitions.

The verifications were performed on a Sparc 10 machine with 64 Mbytes of memory. The average time needed to evaluate each temporal logic formula was less than one minute.

8 Conclusion

Verification techniques allow an early detection of errors in the software life-cycle and significantly contribute to the improvement of program quality.

In this paper, we presented the formal description of the BRP protocol in LOTOS and its verification by model-checking using the CADP (CÆSAR/ALDÉBARAN) protocol engineering toolbox.

The LOTOS descriptions of the BRP protocol and service (i.e., external behaviour) were derived from the corresponding ones written in μCRL by Groote and van de Pol [GvdP93]. The LOTOS descriptions we obtained are quite compact (4 pages and a half of LOTOS instead of 11 pages of MURΦ description language in [HS96]). Although the μCRL descriptions are more compact (1 page), they seem harder to read.

To perform model-checking verifications, we generated the Labelled Transition Systems (LTSs) of the protocol and service descriptions using the CÆSAR and CÆSAR.ADT compilers, by limiting the domains of the protocol parameters. We were able to generate the LTSs for data packets of length 10 and a number of 10 retransmissions in a few minutes on a Sparc 10 machine with 64 Mbytes of memory.

We performed verification by means of two complementary methods: bisimulations (using the ALDÉBARAN tool) and temporal logics (using the XTL prototype model-checker).

In the first approach, we checked that the LTSs corresponding to the BRP protocol and service are branching equivalent. In the second approach, we expressed a set of safety and liveness properties in the ACTL temporal logic and verified them on the LTS of the BRP protocol. In both cases, the average time for each verification was of the order of a few minutes.

It is interesting to compare our results with those obtained using other approaches. The BRP protocol has been already studied in the framework of theorem-proving [GvdP93, HSV94]. These approaches followed the same steps: first, a hand-written correctness proof is constructed, and then it is computer-checked using a theorem prover (like COQ or PVS). An intermediate approach [HS96] consists in combining the advantages of model-checking and theorem-proving, by using a theorem-prover to extract a finite-state abstraction of the system that preserves correctness and then verify it by model-checking.

It is significant to mention that hand-written, computer-checked correctness proofs of the BRP protocol require a large amount of manpower (five man-months in [HSV94] and one man-month in [HS96]), whereas the case-study described in this paper was performed in a few days (less than a week). It is clear that model-checking techniques, although limited to finite-state systems, are simple, efficient, and automated. The future goes certainly through a combination of different methods, in which the model-checking techniques have a significant role to play.

Acknowledgements

Thanks are due to Hubert Garavel, Mihaela Sighireanu, and Laurent Mounier for their suggestions and careful reading of this paper.

A Description of the data structures

We give below the LOTOS description of the data structures used. The types `Boolean` and `NaturalNumber` are taken from the LOTOS libraries given in the ISO standard [ISO88b]. The special comments `(*! constructor *)` are used to explicitly indicate the constructor operators to the CÆSAR.ADT compiler.

```
type DATA is NaturalNumber
   sorts Data
   opns data (*! constructor *) : Nat -> Data
endtype

type PACKET is Boolean, NaturalNumber, DATA
   sorts Packet
   opns nil  (*! constructor *) : -> Packet
        cons (*! constructor *) : Data, Packet -> Packet
        head : Packet -> Data
        tail : Packet -> Packet
        len : Packet -> Nat
        max : -> Nat
        cons_packet : Nat -> Packet
        cons_pack : Nat, Packet -> Packet
   eqns forall P:Packet, D:Data, N:Nat
      ofsort Data
         head (cons (D, P)) = D;
      ofsort Packet
         tail (cons (D, P)) = P;
      ofsort Nat
         len (nil) = 0;
         len (cons (D, P)) = len (P) + 1;
      ofsort Nat
         max = succ (9);
      ofsort Packet
         cons_packet (N) = cons_pack (N, nil);
      ofsort Packet
         cons_pack (0, P) = P;
         cons_pack (succ (N), P) = cons_pack (N, cons (data (succ (N)), P));
endtype

type INDICATION is PACKET
   sorts Ind
   opns I_FST (*! constructor *), I_NOK (*! constructor *),
        I_INC (*! constructor *), I_DK  (*! constructor *),
        I_OK  (*! constructor *)  : -> Ind
```

```
          conf : Packet -> Ind
          ind : Bool, Bool -> Ind
     eqns forall P:Packet, B:Bool
        ofsort Ind
           len (P) == 1 => conf (P) = I_DK;
           conf (P) = I_NOK;
        ofsort Ind
           ind (true, false) = I_FST;
           ind (false, false) = I_INC;
           ind (B, true) = I_OK;
  endtype

  type SIGNAL is
     sorts Sig
     opns START   (*! constructor *), S_READY (*! constructor *),
          RESET   (*! constructor *), R_READY (*! constructor *),
          TIMEOUT (*! constructor *)  : -> Sig
  endtype
```

References

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, January 1988.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CGM$^+$96] Ghassan Chehaibar, Hubert Garavel, Laurent Mounier, Nadia Tawbi, and Ferruccio Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 435–450. IFIP, Chapman & Hall, October 1996. Full version available as INRIA Research Report RR-2958.

[DFH$^+$93] G. Dowek, A. Felty, H. Herbelin, G. P. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Version 5.8. Technical Report, INRIA – Rocquencourt, Rocquencourt, May 1993.

[dMRV92] Jan de Meer, Rudolf Roth, and Son Vuong. Introduction to Algebraic Specifications Based on the Language ACT ONE. *Computer Networks and ISDN Systems*, 23(5):363–392, 1992.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, 1985.

[Fer90] Jean-Claude Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 13(2–3):219–236, May 1990.

[FGK$^+$96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier, and Mihaela Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, August 1996.

[FKM93] Jean-Claude Fernandez, Alain Kerbrat, and Laurent Mounier. Symbolic Equivalence Checking. In C. Courcoubetis, editor, *Proceedings of the 5th Workshop on Computer-Aided Verification (Heraklion, Greece)*, volume 697 of *Lecture Notes in Computer Science*. Springer Verlag, June 1993.

[Gar89] Hubert Garavel. Compilation of LOTOS Abstract Data Types. In Son T. Vuong, editor, *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, pages 147–162. North-Holland, December 1989.

[GP90] J-F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, December 1990.

[GS90] Hubert Garavel and Joseph Sifakis. Compilation and Verification of LOTOS Specifications. In L. Logrippo, R. L. Probert, and H. Ural, editors, *Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada)*, pages 379–394. IFIP, North-Holland, June 1990.

[GT93] Hubert Garavel and Philippe Turlier. CÆSAR.ADT : un compilateur pour les types abstraits algébriques du langage LOTOS. In Rachida Dssouli and Gregor v. Bochmann, editors, *Actes du Colloque Francophone pour l'Ingénierie des Protocoles CFIP'93 (Montréal, Canada)*, 1993.

[Gut77] J. Guttag. Abstract Data Types and the Development of Data Structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[GvdP93] J-F. Groote and J. C. van de Pol. A bounded retransmission protocol for large data packets. Technical Report Logic Group Preprint Series 100, Utrecht University, October 1993.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HS96] K. Havelund and N. Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In *Proceedings of FME '96*, 1996. To appear.

[HSV94] L. Helmink, M. P. A. Sellink, and F. W. Vaandrager. Proof-checking a data link protocol. In H. P. Barendregt and T. Nipkow, editors, *Proceedings of the 1st International Workshop "Types for Proofs and Programs," May 1993 (Nijmegen)*, volume 806 of *Lecture Notes in Computer Science*, pages 127–165, Berlin, 1994. Springer Verlag.

[ISO88a] ISO/IEC. File Transfer, Access and Management. International Standards 8571-*, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, 1988.

[ISO88b] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[ISO89a] ISO/IEC. LOTOS Description of the Session Protocol. Technical Report 9572, International Organization for Standardization — Open Systems Interconnection, Genève, 1989.

[ISO89b] ISO/IEC. LOTOS Description of the Session Service. Technical Report 9571, International Organization for Standardization — Open Systems Interconnection, Genève, 1989.

[ISO92a] ISO/IEC. Approved Algorithms for Message Authentication — Part 2: Message Authenticator Algorithm. International Standard 8731-2, International Organization for Standardization — Banking, Genève, 1992.

[ISO92b] ISO/IEC. Distributed Transaction Processing — Part 3: Protocol Specification. International Standard 10026-3, International Organization for Standardization — Information Technology — Open Systems Interconnection, Genève, 1992.

[ISO92c] ISO/IEC. Formal Description of ISO 8072 in LOTOS. Technical Report 10023, International Organization for Standardization — Telecommunications and Information Exchange between Systems, Genève, 1992.

[ISO92d] ISO/IEC. Formal Description of ISO 8073 (Classes 0, 1, 2, 3) in LOTOS. Technical Report 10024, International Organization for Standardization — Telecommunications and Information Exchange between Systems, Genève, 1992.

[ISO95a] ISO/IEC. LOTOS Description of the CCR Protocol. Technical Report 11590, International Organization for Standardization — Open Systems Interconnection, Genève, 1995.

[ISO95b] ISO/IEC. LOTOS Description of the CCR Service. Technical Report 11589, International Organization for Standardization — Open Systems Interconnection, Genève, 1995.

[Koz83] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[LL95] R. Lai and A. Lo. An Analysis of the ISO FTAM Basic File Protocol Specified in LOTOS. *Australian Computer Journal*, 27(1):1–7, February 1995.

[LT89] N. A. Lynch and M. R. Tuttle. *An introduction to input/output automata.* In *CWI Quarterly*, volume 2, pages 219–246. September 1989.

[Mat94] Radu Mateescu. *Définition et compilation d'un méta-langage pour l'implémentation des logiques temporelles.* DEA, Institut National Polytechnique de Grenoble, June 1994.

[MDI92] R. Melton, D. L. Dill, and C. Norris Ip. Murphi annotated reference manual. Version 2.6. Technical Report, Stanford University, Palo Alto, California, November 1992.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Mun91] Harold B. Munster. LOTOS Specification of the MAA Standard, with an Evaluation of LOTOS. NPL Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[MV93] S. Mauw and G. J. Veltink. *Algebraic Specification of Communication Protocols.* Number 36 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.

[NV90] R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems.* In *Proceedings Ecole de Printemps on Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, 1990.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga - NY, June 1992.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Verlag, March 1981.

[QS83] Jean-Pierre Queille and Joseph Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[Tur93] Kenneth J. Turner, editor. *Using Formal Description Techniques – An Introduction to ESTELLE, LOTOS, and SDL.* John Wiley, 1993.

[vGW89] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399